FlexQueue: Simple and Efficient Priority Queue for System Software

by

Yifan Zhang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2018

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Existing studies of priority queue implementations often focus on improving canonical operations such as `insert` and `deleteMin`, while sacrificing design simplicity and predictable worst-case latency. Design simplicity is sacrificed as the algorithm becomes more and more optimized, taking into account characteristics of the input workload distribution. Predictable worst-case latency is sacrificed when operations such as memory allocation and structural re-organization are deferred until absolutely necessary. While these techniques often yield performance improvement to some degree, it is possible to take a step back and ask a more basic question: is it possible to achieve similar performance while retaining a simple design? By combining techniques such as hierarchical bit-vector and dynamic horizon resizing, all of which are straight-forward in principle, this thesis presents a new priority queue design called FlexQueue, that answers this question with a definitive "yes".

Acknowledgements

I would like to thank all the people who made this thesis possible. In particular, I would like to express my deepest gratitude to my supervisor, Dr. Martin Karsten, for his insight, expertise and wisdom. This thesis would not have been possible without his support.

I would also like to thank my friends Jack and Travis for the time we spent together in Waterloo. These are the most valuable memories that helped me through some of most difficult times.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

A *priority queue* is a data structure that manages a set of key-value pairs, in a way that allows efficient retrieval of the value associated with the smallest key. It is used in many different fields including runtime timer management, discrete event simulation, and memory allocation. In all cases, the efficiency of this data structure is crucial as it is often the most accessed structure in the system [23]. The need for a high performance priority queue thus drives numerous design proposals published over the last few decades.

Among these proposals roughly two research directions can be identified. The first line of research focuses on improving the theoretical bound of canonical operations, such as `insert` and `deleteMin`, which includes tree-based algorithms such as skew heap, binomial heap, and Fibonacci heap. The second direction of research takes into account the characteristics of the input workload and uses this information to dynamically adjust the internal structure. Algorithms in this category often achieve $O(1)$ or amortized $O(1)$ cost for `insert` and `deleteMin` in practice and are generally list-based, such as the two-list procedure [11], timing wheel [57], and calendar queue [19]. However, while improved performance is appreciated, it often comes with trade-offs in terms of complexity and predictability. Novel but complicated design is typically introduced when a heuristic is made better at estimating workload characteristics, and predictability is sacrificed when memory allocation and self-adjustments, in response to a skewed input distribution, are deferred to an unknown point of time in the future. Some of these trade-offs may be more favourable to one type of application than others. For example, in a scheduler for a hard real-time system, a predictable worst-case latency is extremely valuable as this guarantees that operations can be completed within a stringent timeline. In contrast, applications like discrete event simulation can work with amortized complexity and a more relaxed worst-case requirement.

This thesis presents an overview of existing priority queue implementations and discusses their strengths and weaknesses in various applications. Moreover, a different trade-off of complexity and performance is explored through the development and evaluation of a new priority queue design called *FlexQueue* that performs on par with state-of-the-art implementations, while using only a straightforward heuristic and not requiring dynamic memory allocation. This is especially useful, for example, in an operating system kernel or a runtime system. In these systems, intrusive data structures are common because `insert` and `deleteMin` operations do not result in memory allocation or release. To evaluate the performance of FlexQueue, it is benchmarked against several popular tree-based and list-based implementations using the classic *hold* model with *piece-wise* distribution, *normal* distribution, and *triangular* distribution. Results show that FlexQueue outperforms several tree-based implementations including Fibonacci heap and red-black tree. Finally, this thesis modifies a popular network simulator, `ns3`, and replaces its priority queue structure with FlexQueue. A simulator such as `ns3` represents a scenario for which FlexQueue is not optimized because dynamic memory allocation is implied by `ns3`'s source code interface, and amortized overhead is acceptable. Therefore, good performance in this case indicates that FlexQueue is also suitable for more restricted scenarios, for example, when being used inside an operating system kernel.

1.1 Problem Statement

If P is a priority queue, then at minimum the following operations are defined:

- `insert`(P, k, v) Adds the key-value pair (k, v) to P.
- `deleteMin`(P) Returns and deletes the key-value pair with the smallest key.
- `delete`(P, k) If k exists in P, then remove it and its associated value.

In addition to the above, extra operations are defined for priority queues that are used in solving problems such as Dikjstra's shortest path algorithm [26] [30] and minimum spanning tree [53].

- `decreaseKey`(P, k, k') If key k exists in P, then modify it to k'. This is functionally equivalent to `delete` followed by a `insert` and does not change the associated value.

- `merge(`P_1`,`P_2`)` Returns a new priority queue containing all elements from both P_1 and P_2. This is functionally equivalent to a sequence of alternating `deleteMin` and `insert`.

Note that although these compound operations can be implemented using the three basic primitives above, certain designs are more efficient at supporting them. That is, `decreaseKey` can be more efficient compared to a `delete` followed by an `insert`. For this thesis, the primary target scenario is runtime systems in which these operations are not relevant. Therefore, the priority queue design needs to focus only on the basic operations.

As previously mentioned, there are two important trade-offs to consider when designing a priority queue: complexity and predictability. Complexity can be traded for better performance but this trade-off is often not proportional. That is, it can be questionable whether a marginal reduction in time complexity is worth the additional effort required to implement it correctly. There are simple tree-based data structures that offer worst-case complexity of $O(\lg n)$, while more sophisticated designs can achieve $O(\lg \lg n)$ or amortized $O(1)$ time complexity. The proposed priority queue should thus strike a balance between the two, while ensuring worst case is at least $O(\lg n)$. On the other hand, predictability can be traded for better memory utilization. With dynamic memory allocation, only the minimum amount of memory is required to manage queue objects. However, the assumption that memory allocation is $O(1)$ is not always true or reasonable. For example, a system can always run out of memory causing system calls such as `mmap` to fail. Therefore, it can be desirable for the priority queue to allocate required memory ahead of time, while making use of intrusive data structures to eliminate the need for dynamic memory allocation during operations.

The rest of the thesis is organized as follows. Chapter 2 gives an overview of the past research efforts on priority queues and identifies the primary competitors of FlexQueue. Chapter 3 details the design of FlexQueue and the rationale of the choices involved. Chapter 4 presents the experimental setup and evaluation of design. Chapter 5 concludes the thesis.

Chapter 2

Background and Related Work

2.1 Priority Queue and Sorting

The problems of designing an efficient priority queue and sorting integers have received similar treatment in the early literature. In both cases, there is a need to manage a totally ordered set of elements. As a result, solutions to these problems are often very similar. For example, every priority queue can also be used for sorting, by first calling a sequence of `insert` operations followed by a sequence of `deleteMin` operations. If the priority queue supports both `insert` and `deleteMin` in $O(\lg n)$ time, this method results in an $O(n \lg n)$ time sorting algorithm. More recently, Thorup [54] also proved the opposite: that a sorting algorithm with complexity $O(f(n))$ implies that there is a priority queue with $O(\frac{f(n)}{n})$ `insert` and `deleteMin`. In fact, early priority queue designs such as the *binary heap* [62] are originally proposed as a sorting algorithm.

However, despite the homogeneity in pure mathematical terms, there has been no recent priority queue design that starts by building an efficient sorting algorithm, because differences remain between the two problems. In a sorting problem, the data to be processed are often available offline. That is, the problem input is known. A highly optimized sorting algorithm can therefore take advantage of this complete knowledge, and for example apply the median of medians method [13] for pivot selection used in *QuickSort*. On the other hand, the operations of a priority queue are similar to those of an online sorting algorithm, where the data structure must be able to operate efficiently under incomplete knowledge. Furthermore, the output of a sorting algorithm is required to be a list of ordered elements, whereas a priority queue does not necessarily maintain total order on every element at all times. Since the primary operations are `insert` and `deleteMin`, only the largest or

smallest element needs to be readily available. Furthermore, many results in Word-RAM sorting, including Thorup's, introduce hidden constants that are not immediately obvious at a glance. These hidden constants often prevent the theoretical superior techniques from running well in practice.

As a result of such distinctions, studies on these problems became largely independent and research efforts began to diverge. On the one hand, there are continued improvements to the original priority queue design, which focus on the theoretical cost of those primary operations. On the other hand, more recent proposals improve the efficiency by taking advantage of external information, such as knowledge of the input workload.

2.2 Improving the Theoretical Bound

One direction of research into priority queues focuses on general algorithms. That is, algorithms that make no assumption about the types of elements or the system in which they are used. Floyd presents *Treesort* as "Algorithm 113" [27], that uses $O(2n)$ space to sort n elements. This algorithm is essentially a tournament tree that is able to find the largest or smallest elements in $N-1$ comparisons. In this tree, elements are compared pair-wise to determine a "winner", and winners are compared pair-wise repeatedly until one element remains. From this idea, William proposes an implicit version of the tournament tree, known as *binary heap* [62], as the first priority queue implementation that supports `insert` and `deleteMin`. This uses a single n-component array A to store n elements and both operations maintain *heap order*. That is, $A[i] \leq A[2i]$ and $A[i] \leq A[2i+1]$ for all $0 \leq i \leq n-1$. It can be shown that both `insert` and `deleteMin` on the binary heap are $O(\lg n)$ in the worst case, because any out-of-order element $A[j]$ needs at most $O(\lg j)$ swaps with a parent element to restore heap order.

This worst-case bound of $O(\lg n)$ has been improved repeatedly by different researchers. For example, Carlson [20] modifies the implicit heap structure resulting in $O(1)$ `insert`, while `deleteMin` is still $O(\lg n)$. The idea builds on the *binomial heap* [60] in which elements are organized in a forest of trees that are powers of two in size, and two trees in the forest can be merged in constant time if their height is the same.

As mentioned before, any priority queue implementation can be used for sorting by first inserting all elements, and then repeatedly calling `deleteMin`. This implies that a generic priority queue has the same theoretical lower bound as comparison-based sorting algorithms. That is, it is not possible to obtain a better lower bound than $O(\lg n)$ for `insert` and `deleteMin` combined. As such, there is an increased interest in optimizing

other priority queue operations, such as `merge` and `decreaseKey`. A basic binary heap design has the disadvantage that a straightforward `merge` still takes $O(n)$ time. This is improved to $O(\lg n)$ by Crane with the *leftist heap* [25] by maintaining extra *rank* information about each node and its distance to the nearest leaf node. The previously mentioned binomial heap reduces the space usage of leftist heap, since it no longer needs to maintain the same rank information, while achieving amortized $O(1)$ insert. *Skew heap* [50] is later introduced as a self-adjusting leftist heap, and proves that `merge` can be executed in amortized constant time as well. Note that in this case, self-adjusting simply means that the heap itself does not maintain balance information like the leftist heap. Instead, the heap is modified each time it is accessed to maintain balance. Thus, the strength of the skew heap is its amortized cost and not the worst-case cost.

Fredman notes the importance of `decreaseKey` in algorithms such as Dijkstra's single source shortest path. The *Fibonacci heap* [30] is then presented as a solution that supports `decreaseKey` in amortized constant time, improving the solution for the shortest path problem to $O(n \lg n + m)$, where m is the number of edges. Other standard priority queue operations such as `insert` and `deleteMin` are also supported in amortized $O(1)$ and $O(\lg n)$ time, respectively.

So far, those tree-based algorithms do not inherently require dynamic memory allocation, since the metadata of a node can be stored alongside its data using an intrusive data structure. However, they bring increasing complexity as a result of optimizing for the general case. In addition, at least one of `deleteMin` and `insert` still takes $O(\lg n)$ time on average, which is unfavourable for workloads that contain approximately an equal proportion of `insert` and `deleteMin` operations.

More recent priority queues begin to make assumptions about the context of the problem. In discrete event simulation, the *pending event set* contains events that must occur at a specific time in the future. As simulation progresses, these events are removed from the set in temporal order. In this context, keys can be assumed to be non-negative integers that represent the amount of simulation time elapsed since the beginning of the simulation. For example, the van Emde Boas (vEB) tree [14] takes advantage of this assumption, resulting in `insert` and `deleteMin` in $O(\lg \lg N)$ time, where N is the size of the key universe. This is accomplished by recursively dividing the universe N into children of size $\sqrt{n}$. This is a significant improvement over previous results that do not make such assumptions. Johnson [35] builds on this using a non-recursive approach resulting in a $O(\lg \lg D)$ bound, where D is the difference between the smallest and the largest item in the queue. The same idea is used by Anderson to improve radix sort [4]. Thorup [54] modifies this idea again achieving a priority queue that supports `deleteMin` and `insert` in $O(\lg \lg n)$ time while using $O(n2^{\epsilon w})$ space, where n is the number for keys and w is the maximum number of

bits of each key. However, a downside of the vEB tree and its derivatives is that they also require space proportional to the size of the universe N. This $O(N)$ space complexity can be improved to $O(n)$ if hashing is used, allocating a sub-tree for a child only when an element belonging to that sub-tree is first inserted. However, this approach once again requires dynamic memory allocation.

2.3 Timing Wheel and Array-Based Designs

With the advent of data structures like the vEB tree, there has been a shift in research efforts into more specific designs. In comparison to previous tree-based algorithms, those designed for specific applications can make more assumptions that are otherwise impossible. One such assumption is that time is discrete rather than continuous. In other words, time is always represented as an integer. The unit of the integer is typically 1 ns on a modern operating system such as GNU/Linux.

Under these assumptions, Varghese proposes the *timing wheel* [57] using a fixed-size array of list pointers. Each pointer $A[i]$ in the array points to a linked list of timers that are due $i - i_0$ time units in the future, where i_0 represents the current time. Suppose the size of this array is n, and if there is an event that is more than n time units away, then this event is not stored in the array but in a separate sorted linked list. Note that there is no need to sort each $A[i]$ because by definition, all events in a list are due at the same time. In this context, `deleteMin` is less relevant because it is also assumed that each element in the array corresponds to a periodic timer tick, which the operating system must already spend some CPU cycles on bookkeeping. Therefore, it is not useful to be able to look ahead and find an event that is not due immediately. Hence, the timing wheel simply increments i, and processes the list that $A[i]$ points to, if any. This results in a constant time `insert` operation, without amortization.

Brown [19] extends this concept with the *calendar queue*. Instead of letting $A[i]$ represent a single time unit, it can now represent an interval, u, of time units. Effectively, the universe of n future events is partitioned into $m < n$ lists, each of these m lists corresponds to an element in A and is known as a *bucket*. This implies that within a bucket, all events have similar but not necessarily identical timestamps. Thus, there is a choice of implementing the bucket as either a sorted or unsorted list. A sorted bucket means that events are dequeued in the same order as a timing wheel. An unsorted bucket such as a FIFO list means that for each event dequeued at time t, the next event can be up to u time units before or after t, introducing an error proportional to u. Furthermore, the size of A or the number of buckets can be dynamically adjusted as well. In order to limit the

maximum number of events that can be stored in a single bucket, the following method is used:

1. Initially there are two buckets, and $u = 1$.
2. If the total number of events is more than twice the size of A, i.e. the number of buckets, then create a new $A' = A$ with twice the number of buckets and copy all events from A to A'.
3. Similarly, if the number of events is less than half of the size of A, then the A' is created with half the number of buckets.
4. Whenever a new A' is created, the amount of time units each bucket represents is recomputed by calculating the average separation of a fixed number of events at the head of the queue.

This method bounds the average number of times an existing event is copied and prevents any $A[i]$ from eventually having to perform a costly linear search on a large list. If the queue size grows to an exact power of two, then on average each event is copied once. However, in the worst case the queue size grows to one more than a power of two, then all events must be copied once more. More recent variations of the calendar queue propose smarter heuristics on when and how to modify A, but the problem remains that event copying can result in large latency spikes as new events are inserted to ensure they remain in the correct bucket.

Lazy queue [46] and *DSplay queue* [49] are more recent variations that also use a multi-list structure. They do not require resize operations in the same way as other dynamic calendar queues such as the *SNOOPy* queue [52], in which frequent sampling of the input is used to obtain the necessary metrics for deciding when and how to resize. Lazy queue uses an unsorted list for far-away events, and defers sorting to the first `deleteMin` operation. At that time, the bucket width is computed using the minimum and maximum timestamps found in this overflow list, and every event is moved into the appropriate bucket. Then, buckets are emptied sequentially into a splay tree, where they become fully sorted. This eliminates the sampling overhead present in many heuristics-based calendar queue variations. However, as all events in the overflow list must be transferred at the same time, a heavy latency spike is still possible. In contrast, FlexQueue does not rely on intensively sampling the input nor does it require existing events to be relocated each time the bucket width is adjusted. Furthermore, these variations of the original calendar queue do not meet the design objectives because buckets in A are created and destroyed dynamically as the total number of elements grows and shrinks.

2.4 Concurrency

Parallel discrete event simulation (PDES) emerged as an alternative way to cope with a complex system that is far too time consuming to analyze using sequential simulation. PDES attempts to exploit the underlying parallelism in some system models and can drastically speed up a simulation's execution time. However, one of the major challenges as with any concurrency problem is synchronization between execution units. In particular, if simulation events are to be processed in parallel, the implied causality in sequential simulation may not be preserved since the processing order of events is no longer deterministic. In addition, any changes to the global simulation state must be synchronized and this further limits the degree to which the model is parallelizable.

To make use of array-based priority queues in a concurrent application, one could simply apply spin locks either on the queue itself, or on the level of individual elements in the array. Either way, controlling access to the array is analogous to protecting the hash buckets in a hast table. It has been shown [51] that it is sufficient to use a spin lock at the bucket level. This approach only occupies one addition bit per bucket, and is shown to be simpler and faster compared to other techniques such as lock-free queue or reader/writer lock.

2.5 Evaluation Methodology

Regardless of the design choices, it is possible to evaluate tree-based and list-based priority queues using the same methods. For example, *access time* is an intuitive measurement to understand the performance of a priority queue operation. That is, one could measure the time required to perform the most basic operation of `insert` and `deleteMin`. On top of this, the *hold model* has been used for evaluation by nearly every paper that proposes a new priority queue design [45]. In this model, the queue is first populated using a fixed number of items generated by an *increment distribution* $\mathcal{P}_{inc}$. During this time, self-adjusting data structures like the dynamic calendar queue may begin to re-organize according to $\mathcal{P}_{inc}$. After this initial setup phase, the `hold` operation is executed repeatedly, which consists of a `deleteMin` and an `insert`. Subsequent events inserted are generated using the same distribution $\mathcal{P}_{inc}$. The total time for all `hold` operations are then measured and an average access time can be calculated.

This model is a useful representation of a discrete event simulation and can be used to examine the relationship between access time and queue size. However, if the queue is

distribution-aware, either in terms of tree balancing or bucket resizing, then this model can be misleading as the internal queue structure may have changed, even as the total number of elements remains constant. In this situation, it is no longer meaningful to report an average measurement as this does not capture the performance fluctuations as a result of implementation-specific resizing policies. Jones [36] also points out that in addition to the increment distribution, the initial distribution and the resulting queue structure can both have a significant impact on the measured time.

As a generalization of the hold model [45], the Markov model tries to better represent the random nature of event simulation by introducing transition probabilities. In the hold model, a `deleteMin` operation always comes after every `insert`. In a Markov-based model, each `deleteMin` operation has a probability p_1 to transition into the *insert* state where the next operation will be `insert`. Similarly, each `insert` operation has probability p_2 to transition into the *deleteMin* state, making the next operation a `deleteMin`. The result is that there is a random sequence of `insert` and `deleteMin` operations. However, these models tend to mask the internal re-organization of distribution-sensitive queues because they measure the average of a group of temporally sequential operations. Other models exist that are better at exposing this characteristic, and they are often used together with the hold model or the Markov model. The *up-down model* [44] consists of a sequence of `insert` operations followed by a sequence of `deleteMin` operations, making it easy to see the effects of a growing/shrinking queue size and thus the effectiveness of the resize operation.

2.6 Simulator as a Testbed

Aside from benchmark models, which are primarily simulations of real world workloads, it is important to evaluate FlexQueue in a real-world setting. FlexQueue is designed with many restrictions and trade-offs in mind, ones that make it very suitable for use in a operating system kernel where memory footprint should be minimized. However, it would be slow and difficult to assess the effectiveness of FlexQueue's design choices in such an environment, because integration with a kernel is a non-trivial undertaking. Therefore, given priority queue's affinity to discrete event simulation, a network simulator is a natural alternative test platform. A network simulator attempts to model the internal state of a given system such as link speed, congestion, and routing tables. In general, there are two approaches to building a simulator: synchronous and asynchronous. In synchronous or clock-driven simulation, the simulator checks for *events*, or a modification to the current state at each clock tick, giving an impression that time is progressing continuously. In asynchronous or

event-driven simulation, all changes to the simulation state are processed in a first-in-first-out fashion, skipping any clock ticks that have no events attached. These two approaches are not mutually exclusive however, as a system being modelled can exhibit characteristics of both. A event can trigger more events resulting in a complex dependency graph that must be followed to ensure simulation accuracy. Asynchronous simulation represents an interesting test case for FlexQueue because the `deleteMin` operation lends itself naturally to finding the next event to be processed. As an example, Chapter 4 presents modifications to an event-driven simulator `ns3` to use FlexQueue.

Chapter 3

Design of FlexQueue

The original calendar queue as proposed by Brown consists of an N-component array A, storing pointers to N sorted linked lists. Each element in A represents some amount of time units u such that an event with a timestamp of t belongs to a list pointed to by $A[i]$, if $iu \leq t < (i+1)u$. That is, the bucket to which t belongs can be computed as $\lfloor \frac{t}{u} \rfloor$. The horizon of this calendar queue is therefore $h = uN$ time units, and any event that is more than h time units away is stored in the same list as if its timestamp is the remainder of $\frac{t}{h}$. Figure 3.1 illustrates the structure of a calendar queue with five events, and $u = 5, N = 4$. Here, event 30 is considered a distant event because $30 > 5 \cdot 4$, and it is placed in the same bucket as if its timestamp is $30 - 5 \cdot 4 = 10$.

FlexQueue inherits the simplicity of a calendar queue, but differs in a few key areas. First, since a calendar queue uses only a pointer array A to manage buckets, the `deleteMin` operation necessarily has to check each bucket sequentially in order to find the next event. This behaviour is acceptable in cases where the timer facility or operation system is already spending CPU time to process each tick, but becomes inefficient when this per-tick bookkeeping cost can be eliminated, such as in a tick-less runtime where the notion of

```
A[0]:  4        // [20k, 20k+5)
A[1]:  7        // [20k+5, 20k+10)
A[2]:  10,30    // [20k+10, 20k+15)
A[3]:  19       // [20k+15, 20k+20)
```

Figure 3.1: A simple calendar queue with five events

iterating through buckets is nonexistent. FlexQueue adds a *hierarchical bit-vector* on top of A, which contains a summary of the state of A, using one bit for each bucket. A 1-bit indicates that the bucket is non-empty and a 0-bit indicates the bucket is empty. This allows FlexQueue to query the bit vector to find the next non-empty bucket, instead of searching in A directly. With the help of modern CPU instructions, querying the bit-vector can be much faster than searching in A. Second, events outside of the finite horizon represented by A are not stored in the same array as those belonging to the current horizon. Instead, they are stored in a separate overflow list L. This ensures that any non-empty bucket as indicated by the bit-vector does indeed contain the event which should be dequeued next. Finally, FlexQueue uses a different strategy for adjusting the bucket width of A, based on the assumption that regardless of the input distribution, the extent to which a majority of event timestamps deviate from the mean is bounded.

To summarize, in addition to the pointer array from the original calendar queue, FlexQueue also uses:

1. A non-aggressive resizing policy for calculating bucket width.
2. A bit-vector V, that allows the next non-empty bucket in A to be located efficiently;
3. A second pointer array A_2 and bit-vector V_2, that facilitates the non-aggressive resizing policy as explained in Section 3.1.
4. An collection of events L, or the overflow list, for any events beyond the current calendar horizon.

3.1 Pointer Array and Overflow List

Given an uniform input distribution, list-based priority queues, such as the timing wheel and the original calendar queue, are very efficient because the events are uniformly partitioned and distributed into buckets, indexed by their timestamps. In this scenario, there is no bucket that holds a large number of events, nor is there a bucket that holds no events at all. Therefore, the average access time to a bucket is reduced. Furthermore, when the bucket width $u = 1$, as is the case with the original timing wheel, all events in $A[i]$ have identical timestamps and thus the bucket itself can be implemented as an unsorted linked list, instead of a sorted data structure. This further reduces the complexity and average access time to each bucket. However, such assumption is unrealistic as the FlexQueue is

designed to vary the bucket width u periodically, in order to prevent buckets from becoming overpopulated as well as underpopulated.

The pointer array A that FlexQueue uses is a statically allocated N-component array, where N is a pre-determined integer that is a power of two. As a result, the bit-vector V that FlexQueue uses, contains N bits. It is also FlexQueue's goal to attempt to balance the variation in bucket sizes and therefore, it is expected that the majority of buckets will contain some reasonable number of events rather than being empty. This goal further justifies using a static array as the amount of wasted memory is expected to be low. On the other hand, one could easily modify A to be any other randomly accessible data structure, if dynamic memory allocation is not a concern. For example, a hash table can be considered if the total number of events is very small compared to N.

Figure 3.2 illustrates the operation of such a static pointer array whose horizon $h = 20$ does not vary. Solid dots represent events, and are loosely distributed across A. Any non-empty bucket has its corresponding bit in V set to 1. Note here the use of a second pointer array and bit-vector A_2 and V_2. This is done for two reasons. First, using a second pointer array ensures that periodically at least one of the pointer arrays will be empty. If only one pointer array would be used (by treating A_1 as a circular buffer), then it is likely that A_1 is never empty. Resizing A_1 while it is non-empty implies that events need to be relocated to the correct bucket, which can result in a latency spike. On the other hand, using a second pointer array also ensures that the first non-zero bit of V_1 (or V_2, whichever represents the current horizon) always points to the bucket that contains the event with the highest priority. It is explained in Section 3.3.3 why this has the added benefit that searching for the next non-zero bit is faster.

The next question is the type of data structure to be used for each bucket, and whether it should be sorted or unsorted. A sorted data structure is required if the goal is to guarantee all events be dequeued in exact temporal order. This is similar to the behaviour of tree-based algorithms and results in a queue that can perform sorting correctly. An unsorted bucket implies that two events with timestamps e_1 and e_2 dequeued consecutively does not necessarily satisfy $e_1 \leq e_2$. As the bucket width u grows and shrinks, so does the potential distance of events dequeued out-of-order. FlexQueue supports both variants, and can be configured to use either scheme with trivial effort.

It is the responsibility of the overflow list L to hold "outlier" events, so that the remaining events may be stored in the pointer array without introducing too much skew. As a result of this, those events that are not stored in the array may become significantly more skewed compared to the original distribution. Thus, it is logical to implement L using a tree-based algorithm which is insensitive to input distribution.

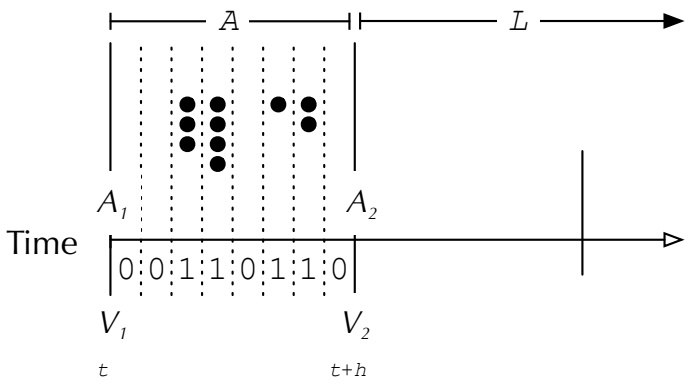


Figure 3.2: Static horizon

3.2 Dynamic Horizon

Given that FlexQueue uses no dynamic memory allocation, this also implies that it cannot allocate memory for new events inserted. Fortunately, intrusive data structures are common in operating system kernels and timer management systems as previously mentioned. In these systems, metadata such as pointers are stored in the same structure as the event itself, using memory previously allocated by a third party such as a user-level program. Furthermore, since changing u is the only dynamic adaptation that does not introduce latency spikes or require memory allocation and expensive event copying, it is important to modify u in a way that ensures a reasonable partition of incoming events. FlexQueue shows that a simple strategy using a dynamic horizon is sufficient to obtain good performance in most cases.

Existing variations of the calendar queue usually compute the number of buckets needed dynamically, in addition to modifying the bucket width. These changes are combined in order to guarantee that subsequent enqueues and dequeues are efficient. More buckets are added when the average number of elements in each bucket is too large. Whenever such a resize occurs, a large number of existing elements may have to be relocated so that they are stored in the correct bucket under the newly computed values. This results in unpredictable memory allocation overhead and a latency spike. FlexQueue uses a fixed-sized bit-vector and pointer array, and varies u, the time spanned by each bucket or the bucket width, instead of adding or removing buckets. This way memory is allocated upfront, and no dynamic allocation is performed during normal queue operations.

Among other things, if the size of L grows too large, this indicates that the current

horizon may be too small and that the pointer array A is not being used efficiently. This is because L is implemented as a tree-based data structure as mentioned before, and if L holds the majority of events, then the expected performance of the queue will approach $O(\lg n)$. In order to ensure such degradation does not occur, the next event to be dequeued must be stored in A with a high probability. In such cases, a resize should be triggered by calculating a new value for u. Ideally, a smooth and even distribution of elements across all buckets is achieved, but this is impossible unless the input distribution itself is uniform. However, an approximation can be made by ensuring the standard deviation of events stored in A does not become too large.

3.2.1 Calculating Horizon Size

Suppose that the input distribution is normal, if all events are stored to A, then a majority of events (approximately 68%) will be concentrated within one standard deviation from the mean. Furthermore, within one standard deviation, the difference between the most and least populated bucket will not be as significant as if the entire population of events are considered. This suggests that u can be adjusted using the sample mean and standard deviation as a guideline. Effectively, A will store all events that fall within a *two-sided truncated* normal distribution. If the input distribution is not normal however, it is not possible to be precise about the degree to which events are centred around the mean, but approximation is still possible through the Chebyshev's inequality [47] which states that:

Chebyshev's Inequality. *If X is a random variable with finite expected value μ and finite non-zero variance σ^2, then for any real number $k > 0$, there is $P(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2}$.*

A direct result of this inequality is that regardless of the input distribution, suppose $k = 1.5$, then at most 44% of all events will be more than 1.5 standard deviations away from the mean. Of course, this is a more relaxed bound compared to one that can be computed if the distribution is known in advance. For example, if $k = 2$, then $P = 25\%$ by this inequality, while a normal distribution with $\mu = 0$ and $\sigma^2 = 1$ has approximately 95% of all values within two standard deviations according to the three-sigma rule [37]. Setting k to a large value guarantees that L cannot hold too many events, which is critical to prevent FlexQueue from degrading to a generic $O(\lg n)$ data structure. At the same time however, this also increases the standard deviation of the truncated distribution that A is meant to store, resulting in an even more skewed distribution in A that L exists to prevent. Clearly, an appropriate value of k is somewhere between these two extremes. If there is prior knowledge of the input distribution, then it is possible to calculate k such that the

size of a bucket is equal to the size of L. For example, assuming the input distribution is normal with mean μ and variance σ^2, then the percentage of events E_A that is stored in A is given by integrating the probability density function of the normal distribution as follows:

$$\begin{aligned} E_A &= \int_{\mu-k\sigma}^{\mu+k\sigma} pdf(x, \mu, \sigma^2)dx \\ &= \int_{\mu-k\sigma}^{\mu+k\sigma} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx \end{aligned}$$

Let $x = y + \mu$, $y = \sqrt{2\sigma^2}z$ and integrating by substitution:

$$\begin{aligned} E_A &= \int_{-\frac{k}{\sqrt{2}}}^{\frac{k}{\sqrt{2}}} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(\sqrt{2\sigma^2}z)^2}{2\sigma^2}} \sqrt{2\sigma^2} dz \\ &= \frac{1}{\sqrt{\pi}} \int_{-\frac{k}{\sqrt{2}}}^{\frac{k}{\sqrt{2}}} e^{-z^2} dz \\ &= \text{erf}\left(\frac{k}{\sqrt{2}}\right) \end{aligned}$$

Therefore, the percentage of events E_H that is stored in L is simply $1 - \text{erf}\left(\frac{k}{\sqrt{2}}\right)$, and the value of k such that

$$\begin{aligned} \frac{\text{erf}\left(\frac{k}{\sqrt{2}}\right)}{N} &= 1 - \text{erf}\left(\frac{k}{\sqrt{2}}\right) \\ \text{erf}\left(\frac{k}{\sqrt{2}}\right) &= \frac{N}{N+1} \end{aligned}$$

guarantees that on average the size of the overflow list is the same as the average bucket size. Here

$$\text{erf}(x) = \frac{1}{\sqrt{\pi}} \int_{-x}^{x} e^{-t^2} dt$$

is a well known non-elementary function, which means that an analytical solution is not possible. An implementation of this function is available in the standard `C++` library that approximates its value with high precision. However, this implementation uses expensive floating point operations that FlexQueue intends to avoid, for example, when being used inside an operating system kernel. Also note that there is no analytical solution for k to

this equation, and an answer can be computed only through methods such as a numerical simulation. This makes it impractical to dynamically adjust k as part of the resizing policy, even with prior knowledge of the input distribution. For this reason, Chapter 4 presents an experiment to determine an appropriate value for k, and leaves this as a configuration parameter, similar to the bucket count N.

In the meantime, both the average and the standard deviation can be approximated efficiently after each `insert` operation, by using an exponential weighted moving average algorithm. Similar to the method used to estimate Round Trip Time (RTT) in TCP [41], this approach has been used successfully in practice and can be implemented efficiently as follows. If the current mean is μ, the current standard deviation is σ and a new event is Δ time units away, then the new estimate $\mu' = \alpha\mu + (1-\alpha)\Delta$, and the new standard deviation $\sigma' = \beta\sigma + (1-\beta)|\Delta - \mu|$ where α and β are the weights given to the current estimate. If α and β can both be written as fractions with powers of two as denominators, then the division operations involved in computing μ' and σ' can be replaced with bit shift operations. For this reason, typically $\alpha = 0.875 = \frac{7}{8}$ and $\beta = 0.25 = \frac{1}{4}$.

Once a new value is calculated for u, this updated estimate can be applied to incoming events by setting `CurrentHorizonMin` to $\mu' - k\sigma'$ or 0, whichever is greater, and `CurrentHorizonMax` to $\mu' + k\sigma'$. To avoid copying existing events and thus introducing a latency spike, a second bit-vector V_2 and pointer array A_2 is used. A_1 and A_2 can have different values for u, as it is updated for one pointer array at a time. Effectively, the actual horizon remains equal to that of a single pointer array. Let u_i denote the bucket width for pointer array A_i, then the correct bucket and pointer array for an event with absolute timestamp t can be calculated as follows in Algorithm 1. This assumes that this event belongs to A, in other words, $t - \textsc{CurrentTime}()$ is between `CurrentHorizonMin` and `CurrentHorizonMax`. The first return value indicates which one of the two bucket arrays the event belongs, and the second return value is the index of the correct bucket.

The bucket width u is updated for an A_i whenever it becomes empty as a result of `deleteMin` operations. This is because applying an update at this point requires no event copying since all events in A_i have just been dequeued. Using the estimated mean μ and standard deviation σ, u_i is updated as follows in Algorithm 2. It is possible that the new horizon value does not divide evenly into N, resulting in a non-integer u. In this case, u is simply rounded up to the nearest integer, and `CurrentHorizonMax` is adjusted accordingly by setting it to `CurrentHorizonMin` $+ u \cdot N$.

Note that a consequence of triggering the resize only after one of A_i becomes empty is that if the current horizon A_i is already holding a large number of events, then Resize is not triggered after all events in A_i are first dequeued. The amount of time this process

Algorithm 1 CalcBucket

```
1: function CalcBucket(t)
2:     interval ← t − CurrentTime()
3:     p ← interval/u_i
4:     if p ≥ N then
5:         p ← (t − (u_i ∗ N))/u_{2−i}
6:         return 2 − i, p
7:     else
8:         return i, p
9:     end if
10: end function
```

takes depend on this arbitrary number, as well as how often incoming events are stored to A_i. If many incoming events are close to `CurrentHorizonMin`, then the number of events in A_i will decrease very slowly. The result is that FlexQueue does not instantly react to input distribution changes. This would only have a severe impact if the input distribution changes drastically as soon as the current horizon comes to an end, which is unlikely.

Figure 3.3 illustrates the idea of a dynamically adjustable horizon. Whenever a horizon has fully elapsed, the empty pointer array has its semantics updated using the estimated mean and standard deviation. This can result in a horizon that does not necessarily begin at the current time. Therefore, in addition to overflowing events, underflowing events (those that are too close to the current time) are also stored in L. To reiterate, the use of a second pointer array creates opportunities where one of them is empty and can be modified without affecting any existing events. Here the effective horizon is represented partially by A_1 and partially by A_2.

Algorithm 2 Resize

```
1: function Resize(μ, σ)
2:     CurrentHorizonMin ← Min(μ − kσ, 0)
3:     CurrentHorizonMax ← μ + kσ
4:     u_i ← ⌈(CurrentHorizonMin + CurrentHorizonMax)/N⌉
5: end function
```

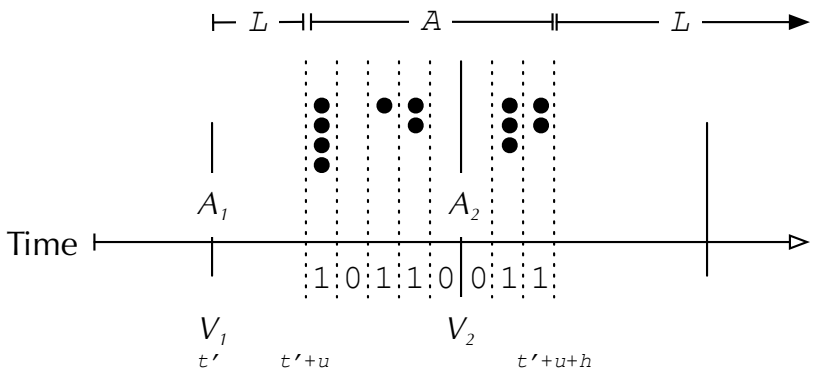


Figure 3.3: Dynamic horizon

3.2.2 Insert

The remaining parts of `Insert` and `deleteMin` are not very different from their counterparts in the calendar queue. When FlexQueue needs to insert an event, it first checks if it belongs to the overflow list by comparing the difference of its timestamp and the current time with the current horizon minimum and maximum, `CurrentHorizonMin` and `CurrentHorizonMax`. If it belongs in A_1 (or A_2), then it calculates the correct bucket in A_1 or A_2 using *interval*, and inserts the event into the corresponding bucket. Finally, FlexQueue updates the current estimate of mean and standard deviation using the moving average method previously mentioned. Pseudocode for `Insert` is presented in Algorithm 3.

As mentioned previously, the values of α and β for the moving average calculation are selected so that floating point operations and integer divisions can be replaced with bit shift. For example $\alpha = 0.875$ means that $\alpha\mu + (1-\alpha)\Delta$ is equivalent to $(7\mu + \Delta) >> 3$.

3.2.3 DeleteMin

When FlexQueue needs to dequeue the next event, it must check both pointer arrays A_1 and A_2 as well as L, and return the event with the highest priority. If the event is inside one of the pointer arrays, then it is removed from the bucket and if no more events remain within the bucket, the corresponding bit in V_1 or V_2 is cleared.

If the current pointer array is A_i but the next event is found in A_{2-i}, then this implies that the current horizon has elapsed and a resize should be triggered. From here, Resize

Algorithm 3 Insert

```
1:  function INSERT(P, t)
2:      interval ← CURRENTTIME() − t
3:      if interval < CurrentHorizonMax & interval ≥ CurrentHorizonMin then
4:          p, i ← CALCBUCKET(t)
5:          BUCKETINSERT(A_i[p], t)
6:          SETBIT(V_i, p)
7:      else
8:          OVERFLOWINSERT(t)
9:      end if
10:     μ ← μα + interval(1 − α)
11:     σ ← σβ + (interval − μ)(1 − β)
12: end function
```

is called and the mean and standard deviation are used to compute the next horizon and its corresponding pointer array A_{2-i}. This process is very efficient because changes to the next horizon are purely semantic: no events are moved and no new buckets are created or existing ones destroyed. Pseudocode for `DeleteMin` is given in Algorithm 4.

3.2.4 Delete

Apart from `insert` and `deleteMin`, `delete` is another operation that is commonly associated with a priority queue. The `delete` operation differs from `deleteMin` in that `deleteMin` only removes the event with the highest priority, while `delete` removes an arbitrary event given its timestamp.

The `delete` operation is important in a timer system such as network protocol timeouts because a large number of timers are created with the assumption that a timeout indicates failure. For example, if a TCP re-transmission timer expires, this means a previous transmission was not successful, potentially due to network congestion causing packets to be dropped. Therefore, most of these timers are actually cancelled during normal operations, before they expire, and are be removed from the queue through `delete`. FlexQueue supports `delete` in the same way `insert` is supported. Using the timestamp, the correct bucket in A_1 or A_2 can be computed with the same algorithm presented in Algorithm 3. Once the correct bucket is located, the appropriate REMOVE function on the bucket is called to remove the event. If the bucket is implemented as a sorted linked list, then REMOVE must iterate through the list until the correct event is found. However, given a

Algorithm 4 DeleteMin

```
 1: function DeleteMin(P)
 2:     pos ← FindFirstSet(V_cur)
 3:     i ← cur
 4:     if pos = INTMAX then
 5:         pos ← FindFirstSet(V_2−cur)
 6:         i ← 2 − cur
 7:     end if
 8:     v ← FirstEvent(A_i[pos])
 9:     h ← FirstEvent(H)
10:     if v < h then
11:         DeleteFirst(A_i[pos])
12:         if A_i[pos] is empty then
13:             BitClear(V_i, pos)
14:         end if
15:         if i not equal cur then
16:             Resize( )
17:         end if
18:         return v
19:     else
20:         return h
21:     end if
22: end function
```

intrusive linkage, there is no need to search in V or A. Assuming the list is doubly linked, the event itself contains sufficient pointer information to perform Remove in $O(1)$ time. Because of the simplicity of this operation, this thesis does not attempt to measure the performance of `delete`, and instead focuses only on `insert` and `deleteMin`.

3.3 Hierarchical Bit Vector

Bit vectors are capable of storing a universe of bits in an extremely compact fashion. With modern CPU support, finding the first set bit in a machine word is very efficient. It is possible to take advantage of this efficiency and construct a bit-vector composed of multiple machine words, such that scanning the bit-vector is equivalent to scanning each machine word sequentially. However, a bit-vector constructed in this fashion is inefficient when

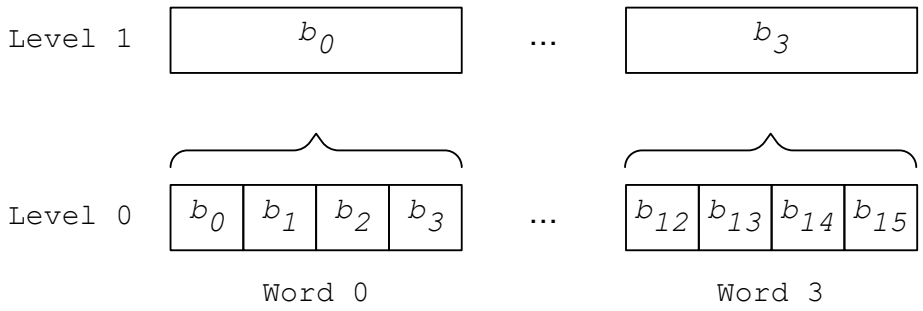


Figure 3.4: A hierarchical bit-vector with capacity $C = 16$ bits

the number of words becomes large, for example, to support a larger universe. One such large universe is the timestamps in event simulation problems, which are typically 64-bit integers. To accommodate this, a bit-vector must have capacity for 2^{64} bits, or 2^{58} words on a 64-bit machine. Scanning these words sequentially is impractical.

Some tree-based priority queues such as the vEB tree use summary vectors to efficiently skip consecutive zero bits during search. Similarly in FlexQueue, the bit-vector is organized hierarchically, with level 0 being the plain bit vector and every level above being a summary vector of the level below. Exactly how many lower level bits are represented by an upper level bit can vary, and this *fanout* factor determines the total number of levels required. For example, a 2^{32} hierarchical bit-vector with a fanout $f = 64$ bits has a total of $L = 6$ levels, 5 of which are summary vectors. In this case, every $f = 64$ bits in a lower level are represented by a single bit in the level above, thus, the total number of levels required is $\lfloor \log_{64} 2^{32} \rfloor = \lfloor 32/6 \rfloor = 6$ levels. Searching in this structure starts from the topmost level and proceeds downward. This scheme requires exactly one word scan on a 64-bit machine at each level and guarantees to find the result in time proportionally to the height of the hierarchical bit-vector. Figure 3.4 shows the organization of a very simple bit-vector with a total of 16 bits and $f = 4$, assuming 4-bit words. At Level 0, all 16 bits are divided into 4 words. This can be stored using an array a with 4 elements, with the least significant bit of the $a[0]$ being b_0, and the most significant bit of $a[3]$ being b_{15}. Since $f = 4$, every four bits in Level 0 is represented by one bit in Level 1, using a total of four bits to represent all 16 bits in Level 0. In this example, two word scans are needed to reach the first set bit, one scan in each level. Generally, searching such a structure is an $O(\lg N)$ operation, where N is the size of the bit universe. Note that N is not the same as the finite horizon of A, since each bit may be used to manage a bucket spanning more than one unit of time.

The next question is whether it is possible to further reduce the latency of searching in this bit-vector. Note that each `bsf` scan instruction can scan at most one machine word

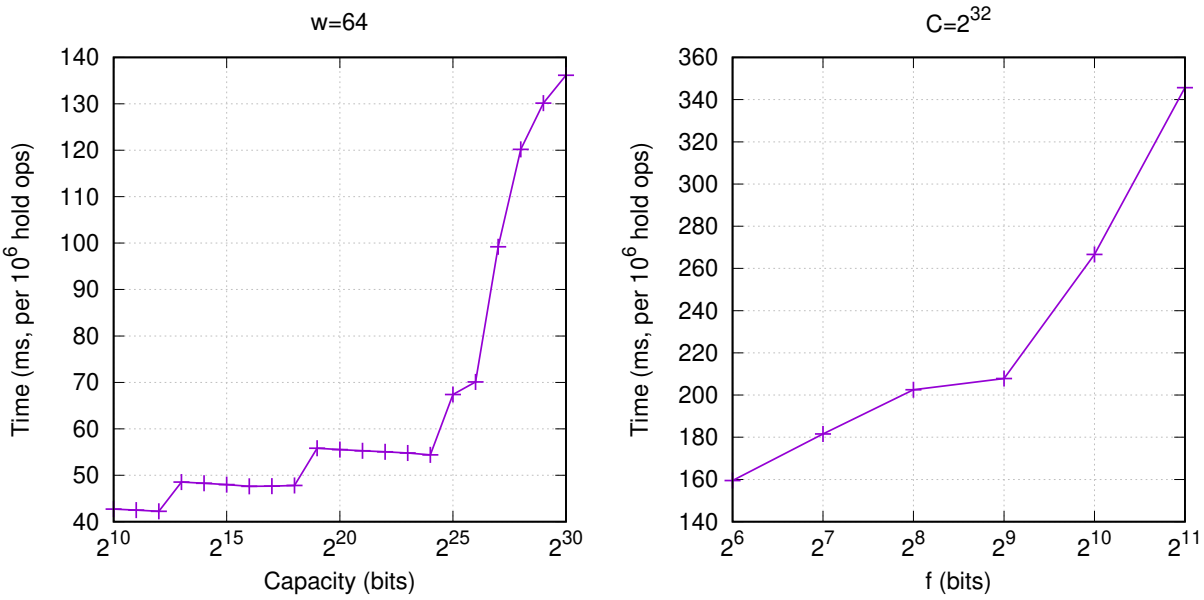
(typically $W = 64$ bits) and modern CPUs typically have a cache line size of 64 bytes (512 bits). This implies that scanning a level when $f = 64$ loads not just the word required for `bsf`, but also an additional 56 bytes from memory. These extra bytes are not needed for searching in the current level, and therefore, cache loads are not being used in the most efficient manner for any $f < 512$. By increasing f, the number of extraneous bytes loaded decreases and so does the number of levels. Thus the height of the bit-vector will be reduced as well, without incurring extra cache loads. A bit vector with 2^{32} bits now has four levels, if f is increased to 256 from 64. This improved cache utilization may result in a faster search operation. However, as the number of cache loads remain constant per level, the downside is that this also increases the number of scans required per level, from one $f = 64$ to four at $f = 256$. If a hierarchical bit-vector has T bits and the fanout is f, then it has $L = \lfloor \frac{\lg T}{\lg f} \rfloor$ levels. Clearly, if $f' = f^2$, then $L' = \frac{1}{2}L$. Since $n = \lceil f/W \rceil$ scans are needed per level, this changes the number of searches required to

$$L'n' = \frac{1}{2}L \cdot nf = \frac{f}{2}Ln.$$

Clearly, as long as $f > 2$, this results in a larger number of searches overall. Figure 3.5 shows the effect of fanout selection, with various levels. In Figure 3.5a, the horizontal axis represents the total number of bits in the bit-vector, while the vertical axis represents the latency of one million hold operations, in microseconds. It is clearly visible that there is a latency increase at 2^{13}, 2^{19}, and 2^{25}, while the latency stays relatively flat for each interval in-between. These values correspond to when the number of levels of the bit-vectors increases by 1 and illustrates the effect on latency when the number of levels is increased. In Figure 3.5b, the total number of bits is fixed, and the fanout f is varied from 64 to 2048. This illustrates the effect of decreasing the number of levels, while making each level more expensive to search. The latency of finding the first set bit increase more drastically after $f = 2^9$, since this means that a summary vector now occupies at least two cache lines rather than one, when $f < 2^9$. Thus, for the remainder of the thesis, it is assumed that $f = 64$.

3.3.1 SetBit Operation

A bit in V can be uniquely identified using three subscripts: the i-th level, the k-th word, and the r-th bit. Setting a bit is performed bottom-up: setting the bit in Level i, then setting the $\lfloor \frac{i}{fanout} \rfloor$-th bit in Level $i+1$, and then updating the summary vectors repeatedly until the top-most level is reached. This requires exactly L iterations, although it is possible to stop as soon as $V[i][k][r] = 1$, since every level thereafter must already be 1.



(a) as a function of C (b) as a function of f

Figure 3.5: Impact of fanout factor f on hold time

Algorithm 5 BitSet

```
1: function BitSet(V, L, k)
2:     for i ← 0, L − 1 do
3:         r ← k mod f
4:         k ← k/B
5:         V[i][k][r] = 1
6:     end for
7: end function
```

3.3.2 ClearBit Operation

Setting a bit in V to zero is slightly more involved, because in order to clear the corresponding bit in Level $i+1$, the current word must entirely consist of zero bits. This requires a branch and the loop must terminate as soon as the current word is no longer zero. Similar to `BitSet`, `BitClear` also requires at most L iterations.

Algorithm 6 BitClear

```
1: function BitClear(V, L, k)
2:     for i ← 0, L − 1 do
3:         r ← k mod f
4:         k ← k/B
5:         V[i][k][r] = 0
6:         if V[i][k] not 0 then
7:             return
8:         end if
9:     end for
10: end function
```

3.3.3 FindFirstSet Operation

Finding the first non-zero bit is also very straightforward. Starting at Level L, which by definition has at most one word, find the first non-zero bit at position p, then the next word to scan is simply the p-th word in Level $L-1$. This is repeated until level 0 is reached, at which point the final position can be calculated.

WordScan is a wrapper function that calls the built-in macro `__builtin_ffs` in GNU `gcc`, which depending on the implementation and the architecture, uses `bsf` on x86, or an functionally equivalent software implementation if `bsf` is not supported in hardware. According to the GNU `gcc` documentation, if the return value of this built-in macro is zero, it indicates that the input word is empty. In this case, f is returned as a signal that no one-bit is found. Otherwise, the return value s is the index of the least significant one-bit of W, starting at 0.

Variations of FindFirstSet are possible, for example, if one wishes to begin the search not from the least significant end, but rather from some other non-zero position p. In this case, a simple top-down scan as presented in Algorithm 7 does not work, since finding the first non-zero bit using this algorithm relies on the principle that the first non-zero bit

Algorithm 7 FindFirstSet

```
1: function FindFirstSet(V, L)
2:     p ← WordScan(V[L − 1][0])
3:     if p = f then
4:         return INTMAX
5:     end if
6:     for i ← L − 2, 0 do
7:         l ← WordScan(V[i][p])
8:         p ← f ∗ p + l
9:     end for
10:    return p
11: end function
```

Algorithm 8 WordScan

```
1: function WordScan(W)
2:     s ← __BUILTIN_FFS(W)
3:     if s = 0 then
4:         return f
5:     else
6:         return s − 1
7:     end if
8: end function
```

in Level i is represented also by the first non-zero bit on Level $i + 1$. The same however does not apply to an arbitrary p. At minimum, the Level 0 word in which bit p is stored (word $\lfloor \frac{p}{f} \rfloor$) must be searched since there is no information elsewhere on whether there are multiple 1-bits in a word. If there is no one-bit after bit p on Level 0, then the bits on Level 1 can be used to the find the next non-zero word, by searching word $\lfloor \frac{p}{f^2} \rfloor$ on Level 1. If the next one-bit is far away from p, then this process may repeat until Level L is reached, after which a top-down search can be performed in the same way as Algorithm 7. This variation requires at most $2L$ iterations and is slower than when p can be assumed to be zero.

In any case, V is only responsible for the meta-information, while the events themselves are managed by the pointer array A. After the first non-empty bucket is located efficiently, the correct event can be removed from that bucket.

Chapter 4

Benchmarks

As previously mentioned, the primary operations on a priority queue are `insert` and `deleteMin`, which is the focus of evaluation in this chapter. While other common operations, such as `decreaseKey` and `delete`, are not directly evaluated, the methods and results presented in this chapter allow the observation of FlexQueue's performance under the most common workloads.

For the purpose of clarity, a preliminary set of experiments are conducted on the individual components of FlexQueue: the bit-vector and the bucket structure. The first part of these experiments is intended to measure the average access time of a bit-vector and a linked-list. The linked-list is considered to be the simplest and fastest data structure that may be used to implement buckets, thus, this comparison shows the maximum overhead of adding a bit-vector on top of the pointer array. The second part of these experiments focuses on the bucket itself. Specifically, these experiments measure the difference between using an intrusive and non-intrusive data structure.

Section 4.3 present experiments that evaluate the effectiveness of FlexQueue's resizing policy. FlexQueue's hold time is first measured when the horizon is not permitted to vary dynamically. The results are then contrasted with those when the resizing policy is employed. Next, the performance of `insert` and `deleteMin` are compared to Fibonacci heap, red-black tree, binomial heap, and the original calendar queue. Implementations of these queues are freely available in third-party libraries such as `boost` except for the calendar queue, and they are selected for the following reasons:

1. Fibonacci queue claims amortized $O(1)$ time for not just `insert` and `deleteMin`, but also `delete` and `decreaseKey`. This implies that it has an average case complexity that is theoretically comparable to a calendar queue under optimal conditions.

2. Red-black tree is the implementation used by the `GNU C++` library for the `map` and `set` data structure. It is readily available for any `C++` program and thus it can be considered a default choice when a priority queue is needed. It is also the priority queue used by the popular network simulator `ns3`. For simplicity, this thesis refers to this as the set queue.

3. Binomial queue is the one of the earliest designs of priority queue, and it claims to have favourable worst-case performance than other tree-based algorithms.

4. The original calendar queue performs well if the input distribution is uniform and if the number of queue items does not fluctuate significantly. However, costly event copying and poor resizing policy are two flaws that are immediately obvious when compared with FlexQueue. This serves as a baseline for other calendar queue variations.

4.1 Experiment Design

One of the primary objectives of FlexQueue is to achieve comparable performance to existing priority queue implementations, while not requiring complex heuristics and dynamic memory allocation. Other objectives are arguably less relevant, if FlexQueue does not at least perform comparatively in an optimal setup, such as uniform input distribution. In this distribution, the number of events in each bucket remain consistent across the entire horizon. Thus, the average access time is minimized. More importantly, performance is critical especially when the queue is intended to be useful in operating system kernels and timer management systems. This section focuses on synthetic benchmarks and access patterns, rather than real workloads. These benchmarks are specifically crafted to stress FlexQueue's implementation in a particular way by allowing experiment parameters to be easily controlled.

Each benchmark consists of three stages: model selection, initialization, and simulation. During the model selection stage, one of the three experiment models are selected as the harness that drives the remaining two stages. Each model represents a different set of criteria for performance evaluation. For example, the hold model measures the average latency for a pair of `insert` and `deleteMin` operations under a fixed queue size, while the Markov model measures the average access time of a single operation which may be `insert` with probability p and `deleteMin` with probability $1 - p$. During the initialization stage, an increment distribution $\mathcal{P}_{inc}$ is selected that is used to populate the queue being evaluated until the number of events in the queue reaches the specified size. Finally, during

the simulation stage, a large number of `insert` and `deleteMin` operations are executed on the queue and the relevant performance metrics are measured and reported.

4.1.1 Hold Model

The most commonly used model to measure a priority queue's performance is the hold model. The objective of this model is to measure the time of `insert` and `deleteMin` combined. This simulates a scenario where the number of events stored in a queue converges to a steady state value. A plausible reason that this model is widely used is that most tree-based implementations have time complexities that are proportional to the size of the queue. It is therefore convenient to use a model where this variable can be easily controlled.

4.1.2 Markov Model

In a discrete event simulator, simulation typically begins by generating a large number of events, causing the priority queue size to steadily increase, and then finishes by eventually deleting every event from the queue. A Markov model with parameters p_0 and p_1 generalizes the classic hold model. By decoupling `insert` and `deleteMin`, such a model allows `insert` to precede a second `insert` with probability p_0, and `deleteMin` to precede a second `deleteMin` with probability p_1. Clearly, the hold model is a special case where $p_0 = p_1 = 0$. This results in a slightly more general approximation of a real-world workload, since it is not likely that each `deleteMin` produces exactly one `insert` as is the case with the hold model.

4.1.3 Transient Model

Distribution-sensitive priority queues, which in general include all variations of the calendar queue, modify their internal structure as a response to shifting input distribution. Therefore, assuming the input distribution remains stable, it is expected that these queues reach a steady state after some initial period of time. On the other hand, tree-based priority queues are often insensitive to input distribution, and the latency of the `insert` and `deleteMin` operation does not typically fluctuate as the distribution shifts. In both the Markov and the hold model, measurements made during and across distribution shifts are reported as a single average, which does not reflect the transient state of the queue, but also does not reflect a meaningful steady state. While this is not an issue for tree-based implementations, for FlexQueue there may be a significant difference between the observed

performance at the beginning of an experiment and after this initial period of time. For these types of queues including FlexQueue, it is helpful to understand if and when the data structure initiates this transient behaviour, and for how long it persists. This can be accomplished by taking the average of a smaller set of consecutive hold operations. For example, instead of reporting the average of one million hold operations, these can be divided into smaller groups of 10,000, and reporting 100 data points. In other words, by measuring over a smaller interval, it is possible to observe, at a finer granularity, the change in latency as a function of time.

There is another type of transient behaviour that is best described as the warm-up period. When a benchmark program is first loaded into memory and execution begins, CPU cache lines are not yet populated, or contain data from other programs running on the same CPU that may need to be evicted. Once the benchmark has executed the `deleteMin-insert` loop a few times, it is more likely that measurements will be protected from such effects.

4.1.4 Increment Distributions

The increment distribution $\mathcal{P}_{inc}$ is used to generated event timestamps, or priorities. Given the timestamp t of a previously expired event, $\mathcal{P}_{inc}$ produces an non-negative integer Δ, according to some pre-determine distribution parameters. The timestamp of the new event is simply $t + \Delta$. Different distribution parameters can result in vastly distinct access patterns.

A common distribution used in bench-marking is the uniform distribution $U(a, b)$. This distribution produces values between real numbers a and b, each with equal probability of $\frac{1}{b-a}$. This is a useful distribution because in addition to its simplicity, if a and b are assigned the lower and upper bound respectively of the timestamps domain, then this distribution is effectively a random number generator. As a result, the average case latency of a priority queue can be measured. However, such a simple distribution is insufficient when it comes to modelling a real workload. A normal distribution $N(\mu, \sigma^2)$ with mean μ and standard deviation σ comes much closer to a real workload than uniform distribution. As stated in previous sections, the majority of events are centred around the mean in a normal distribution, and events with larger or smaller timestamps are exponentially less probable. This is useful in modelling workloads that exhibit temporal locality. That is, an simulation event is more likely to trigger events that are closer to itself than to trigger events that are more distant into the future.

While both uniform and normal distribution are useful models for workloads with dis-

tinct characteristics, they are insufficient when the workloads becomes more complex. A complex workload does not necessarily have a clear cluster, nor are its events uniformly distributed. A triangular distribution $\text{Tri}(a, b)$ generates values distributed over the interval $[a, b]$, such that the probability increases linearly from a to b. This distribution is useful because unlike the normal distribution, there is clear cluster and thus can represent a workload with mixed characteristics.

4.2 Setup and Implementation

The benchmarks are performed on the following platform:

- Intel Xeon E5-4610 8-core CPU with Hyper-Threading turned on
- 32 KiB L1 data cache, 256 KiB L2 cache, and 16 MiB L3 cache
- Linux v4.13.0-38, gcc-7.2.0
- 256 GB memory

Regardless of the models used, the core logic of the experiment harness remains largely identical. The harness is divided in to two major modules: harness and test case.

4.2.1 Harness

This module is responsible for operations such as collecting input parameters, measure run time of test cases, and printing the output to a white space separated row suitable for use in a graphing utility such as `gnuplot`. It contains only the most basic logic required to start a benchmark. Listing 4.1 shows the basic operations of the harness. Note that each data point plotted is the average of 20 samples, as indicated by the `num_trials` variable in Listing 4.1. Results in Section 4.3 show that majority of measurements have a relative standard error of less than 2.3%. That is, the true mean is likely to be within 2.3% of the sample mean that is shown in subsequent figures. Occasional outliers exist with relative standard errors up to 9%, these are noted where appropriate.

```
int main() {
  const int num_trials = 20;
  const int num_repeat = 1000000;
```

```
  TrialRun time;

  // create test case from supplied name
  TestCase *testCase = tcMap.at(XSTR(TEST_NAME))();

  testCase->setUp();
  for (int j = 0; j < num_trials; j++) {
    auto start = chrono::steady_clock::now();
    testCase->run(num_repeat);
    auto end = chrono::steady_clock::now();
    auto elapsed = chrono::duration_cast<chrono::microseconds>(end - start);
    time.addResult(elapsed.count());
  }
  testCase->tearDown();
  delete testCase;

  cout << time.mean() << "\t" << time.stddev() << endl;
}
```

Listing 4.1: Operations of the harness

4.2.2 Test Cases

Test cases are collections of C++ classes that decide the content of a benchmark, including the model used, the queue tested and the input distribution used. Listing 4.2 shows the operation of a test case that implements the hold model using the normal distribution to test FlexQueue.

```
class FlexQueueTestCase {
  size_t _uid_counter = 0;
  FlexQueue _queue;
  Distribution *_dist;

  public:
  FlexQueueTestCase() {
    _dist = distMap[XSTR(DISTRIBUTION)]();
  }

  // populate queue to desired size
```

```
  virtual void setUp() {
    SetCurrentTime(0);
    for (size_t j = 0; j < QueueSize; j++) {
      Event e{};
      e.key.m_ts = _dist->next();
      e.key.m_uid = _uid_counter++;
      _queue.Insert(e);
    }
  }

  // begin the hold operation
  virtual void run(int repeat) {
    for (int i=0; i<repeat; i++) {
      Event e = _queue.RemoveNext();
      SetCurrentTime(e.key.m_ts);
      size_t n = _dist->next();
      e.key.m_ts = n + GetCurrentTime(); // a random interval + now
      e.key.m_uid = _uid_counter++;
      _queue.Insert(e);
    }
  }
};
```

Listing 4.2: A FlexQueue test case

While piece-wise and normal distribution's implementation are both available directly through the `C++` standard library, the triangular distribution is not. Nevertheless, it is simple to derive an equation from the definition of triangular distribution: $\mathrm{Tri}(a, b)$ can be generated by $a + (b - a)\sqrt{\texttt{rand()}}$, where `rand()` generates uniformly distributed real numbers from $[0, 1]$. Figure 4.1 show the values generated by a triangular distribution with $a = 10^6$ and $b = 10^8$. Assuming that event timestamps are recorded in nanoseconds, this represents a synthetic workload where there are more events 100 ms away than there are events 1 ms away.

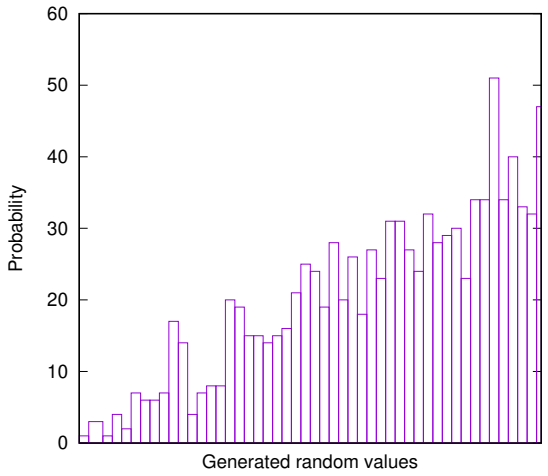


Figure 4.1: Frequency histogram of Tri($10^6, 10^8$)

4.3 Results

4.3.1 Preliminaries

Intuitively, setting a bit in a machine word is very fast. However, in a hierarchical bit-vector this change must propagate to several words at higher levels. It is not obvious if and how much overhead such an operation introduces, compared to inserting and removing from a linked list, which is generally considered to be $O(1)$. Thus, the objective of this experiment is to determine the cost of operations on the hierarchical bit-vector, relative to operations on the pointer array A.

In this experiment, an N-component array is created and its elements are accessed both sequentially and using a randomized pointer chasing method. Such a method effectively disables memory prefetching, and represents a maximum difference between accessing a linked-list and accessing a hierarchical bit vector.

For the linked-list, the array is pre-allocated to store nodes that are later appended to the linked-list. In the sequential experiment, each array element is accessed sequentially, and linked to the previous node using standard linked-list operations. In the randomized pointer-chasing experiment, before $A[i]$ is appended to the linked list, the value of $A[i]$ indicates the index of the next element in A that should be inserted. Indices are generated by first writing sequential integers starting from 0 into A, then shuffled using the `C++` standard library function `std::shuffle`.

Capacity (bits)	Memory Used (bytes)	Fits Into
64	16	L1
2^{10}	272	L1
2^{12}	1,040	L1
2^{20}	266,320	L1 and L2
2^{32}	1,090,785,360	Does not fit

Table 4.1: Total memory used by bit-vectors V_1 and V_2

Similarly for the bit-vector, the sequential experiment starts from the least significant bit, change its value to 1, then moves on to the next least significant bit and so on. In the randomized experiment however, there are no arrays or pointers for the bit-vector to chase, therefore, instead of creating a dedicated array, a random bit is set for each insertion.

Figure 4.2 shows the results of inserting sequentially into a linked list versus a bit-vector. The total number of items inserted is represented on the horizontal axis and the latency of insertion is represented on the vertical axis. Multiples of 2^{18} are selected because FlexQueue's bit vectors fit in to the L1 and L2 cache of the test machine's CPU. Refer to Section 4.3.2 and Table 4.1 for more details.

For the bit-vector, the horizontal axis represents the capacity, rather than the number of bits set. For sequential insertion, the cost of these two types of data structures are very close, suggesting that operations on a bit-vector are nearly as costly as operation on the linked list itself. However, it is important to note that for the linked list experiment, items are taken from a pre-allocated array sequentially. Therefore, it benefits significantly from cache locality and is not indicative of a real-world workload.

Indeed, the right side of Figure 4.2 shows the result of random insertion, where much of this benefit for the linked list is removed. Consequently, linked-list operations become very expensive on average due to cache misses. In contrast, the hierarchical bit-vector only uses one bit per element whereas the linked-list uses 64 times more memory, on a 64-bit machine. Such space efficiency implies that the CPU cache can hold more elements from the bit-vector compared to the linked list. As a result, the number of cache misses are reduced significantly when accessing the bit-vector resulting in faster average access time.

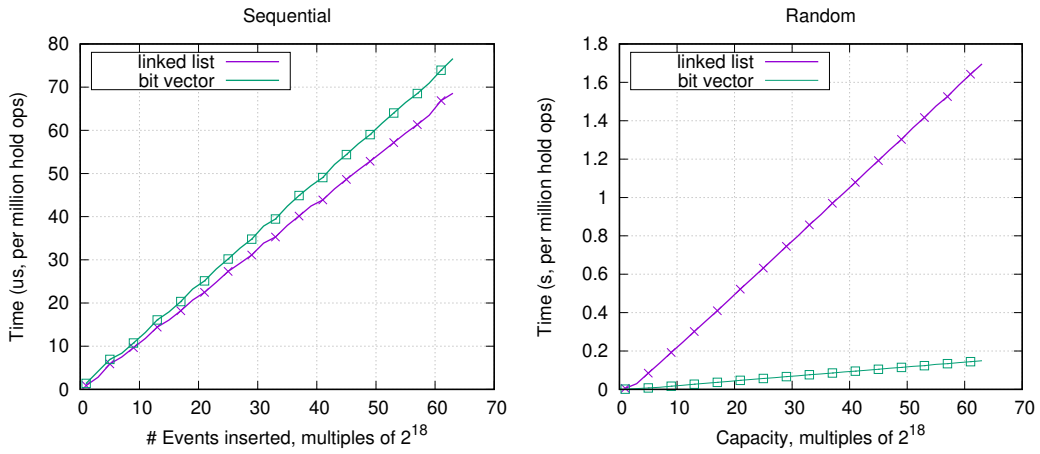
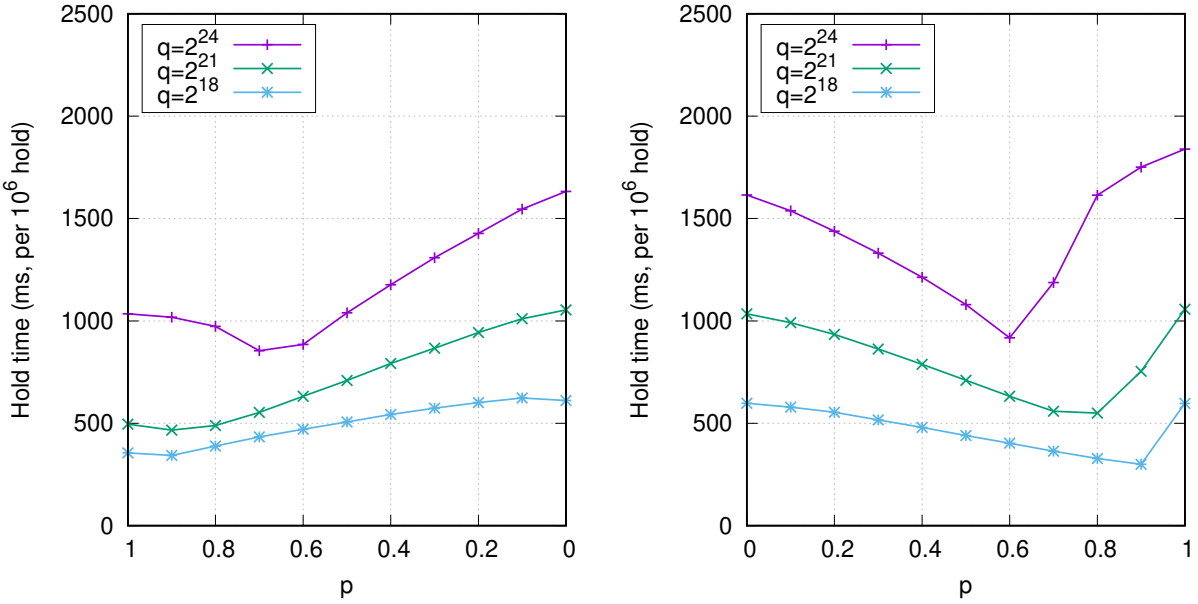


Figure 4.2: Cost of Bit Vector Operations

4.3.2 Static Horizon

This is a scenario in which the bucket width u is not allowed to change. To demonstrate the consequence of not responding to changes in input distribution, the increment distribution $\mathcal{P}_{inc}$ used is *piece-wise*, such that $x\%$ of all events are distributed evenly across a fixed horizon, while the remaining $(100 - x)\%$ are outside of the horizon and thus stored in the overflow list. Therefore, this represents the best case performance attainable by FlexQueue when $x = 100$. In all cases, the number of buckets is fixed at 2^{20}. This number is the largest power of two such that the two bit-vectors V_1 and V_2 used by FlexQueue fit into L1 and L2 cache of the CPU in the test environment. Table 4.1 shows the amount of memory used by FlexQueue's bit-vectors at various configurations.

Figure 4.3 shows the performance of FlexQueue against the set queue under various setups. Note that on the horizontal axis in Figure 4.3a, p decreases from 1 to 0, whereas it increases from 0 to 1 in Figure 4.3b. This contrast emphasizes the fact that when $p = 0$, the static setup stores all events using the overflow list, which results in similar performance as SetQueue in Figure 4.3b. In addition, when the queue size is relatively small at 2^{18} and as the percentage of events in A approaches 0%, the average hold time decreases slightly. This implies that set queue is slightly more efficient at managing small sets of events. In contrast, as the queue size increases, it becomes more expensive to use set queue, while the cost of FlexQueue does not increase as significantly. It is interesting to note that the set queue also responds to distribution shift. Upon investigation, this is because the piece-wise distribution used in this experiment uses the interval $[0, 2^{20}]$ and $(2^{20}, 2^{32} - 1]$ with



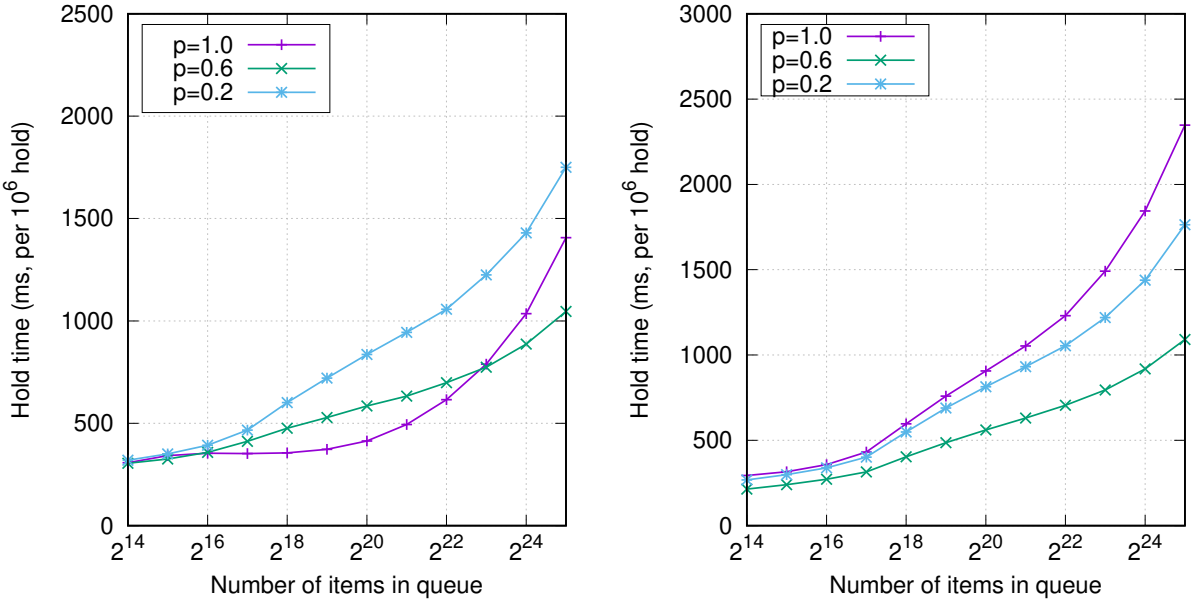
(a) FlexQueue static

(b) SetQueue

Figure 4.3: Time vs. p under various q

probability p and $1 - p$ respectively. That is, events outside of the horizon is distributed over a much larger interval than those within the horizon. As the total number of events outside the horizon grows, they occupy more and more non-leaf nodes in a balanced binary tree, such as the set queue. As a result, the remaining events within the horizon are pushed into a small sub-tree, while the rest of the events belong to a much larger sub-tree, with a higher access time. Since these events are always dequeued before the others, effectively the working size, or height, of the whole tree is reduced. However, even with this unintended advantage, set queue does not outperform FlexQueue. This is especially obvious when the queue size is large. When $p = 0$, both figures are effectively representing the same data structure, and thus converge. Note that the measurements for set queue have relatively large standard errors for $0.6 < p \leq 1$, this is possibly because as p approaches 0.5, it becomes more likely that both sub-trees are accessed. Since the access time of these two sub-trees differ, measurements across these ranges of p values are no longer stable.

Figure 4.4 shows that as long as most events are stored within the pointer array, the average hold time does not noticeably increase as queue size increases. However, when the number of events is greater than the number of buckets, further increasing the queue size does still increase the average access time of FlexQueue. Intuitively, since $N = 20$, on



(a) FlexQueue static

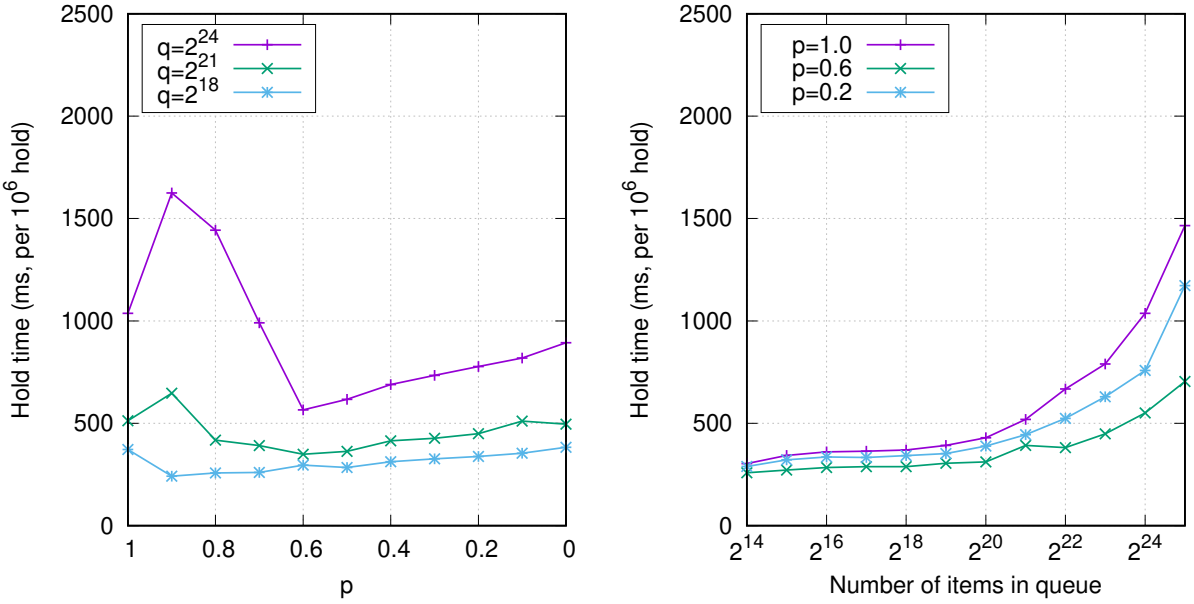
(b) SetQueue

Figure 4.4: Time vs. queue size under various p

average each bucket, implemented with a set queue as well, holds a constant factor of 2^{20} fewer events compared to the single set queue. When the number of events is less than 2^{20}, there exists one or more empty bucket. Thus, increasing the queue size merely fills those empty buckets, without increasing the average latency.

4.3.3 Dynamic Horizon

Figure 4.5 shows the result of enabling the resizing policy. When the number of events is 2^{18} and $p = 0$, the resizing policy estimates the mean and standard deviation of $\mathcal{P}_{inc}$ and attempts to resize to a smaller horizon. However, since the bucket width can be no less than 1, FlexQueue does not make any changes to the horizon size. Notice that compared to the static scenario, the hold time no longer briefly increases up to $p = 0.8$. This is the result of the resizing policy modifying the boundaries of the horizon. As more events are shifted from A to L, the mean of event timestamps increases. FlexQueue senses such a shift and "tracks" it by increasing the lower bound of the horizon. As a result, hold time remains relatively flat from $p = 1$ to $p = 0$.



(a) as a function of p

(b) as a function of q

Figure 4.5: Effect of dynamic horizon

Notice that since the resizing policy is designed to minimize the negatives of a biased input distribution, it has no effect on nor is it impacted by the queue size. As Figure 4.5b shows, employing a dynamic horizon does not improve FlexQueue's efficiency when every bucket needs to store more events on average.

4.3.4 Determining the Value of k

Briefly recall k from Section 3.2.1, the appropriate value of k may differ depending on the input distribution. For example, for a highly clustered distribution such as a normal distribution with small variance, k may need to be large in order to effectively make use of the buckets in A. On the other hand, a large k increases the difference between the size of the largest and smallest bucket in A. This causes the distribution of events in A to approach the actual workload distribution, which is presumably skewed. Section 3.2.1 states that the following must be satisfied in order for L and each bucket in A to have the same size:

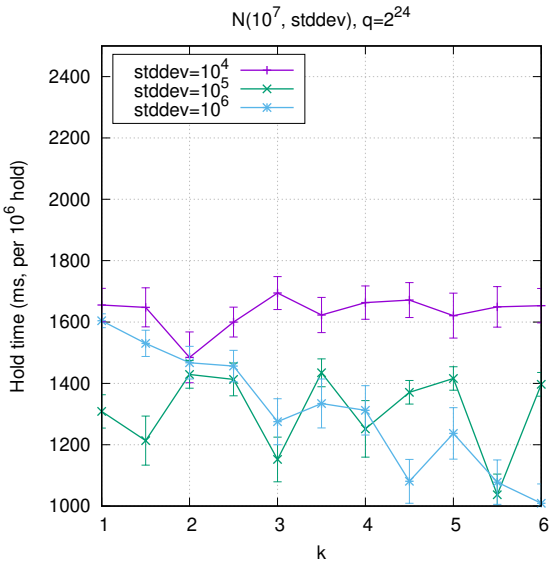


Figure 4.6: Effect of k

$$\operatorname{erf}\left(\frac{k}{\sqrt{2}}\right) = \frac{N}{N+1}.$$

Since `erf` is an increasing function, it is possible to approximate the value of k using binary search by setting lower bound to 1 and upper bound to a sufficiently large value such as 10^8. When $N = 2^{20}$ as is the case with previous experiments, k is estimated to be approximately 4.90096. Figure 4.6 shows the effect of k on a normal distribution benchmark with various standard deviations. Clearly, when the majority of events are clustered as is the case with $\sigma = 10000$, varying k does not help. This is because when k is small, a large percentage of events reside in L; when k is large, a small number of buckets in A hold a majority of the events. Since bucket is also implemented using a set queue similar to L, regardless of k, in both cases the average latency is dominated by the performance of the set queue.

On the other hand, when σ is large, i.e. the events are more scattered, then increasing k may marginally reduce the latency, by bringing more events into every bucket in A instead of just a few when then events are tightly clustered. There is no single value of k that is optimal for all distributions, and this experiment suggests that a large default value such as 6 is safer than a small value such as 2. While a large k does not necessarily reduce access time, it is reasonable to assume that it does not increase access time either.

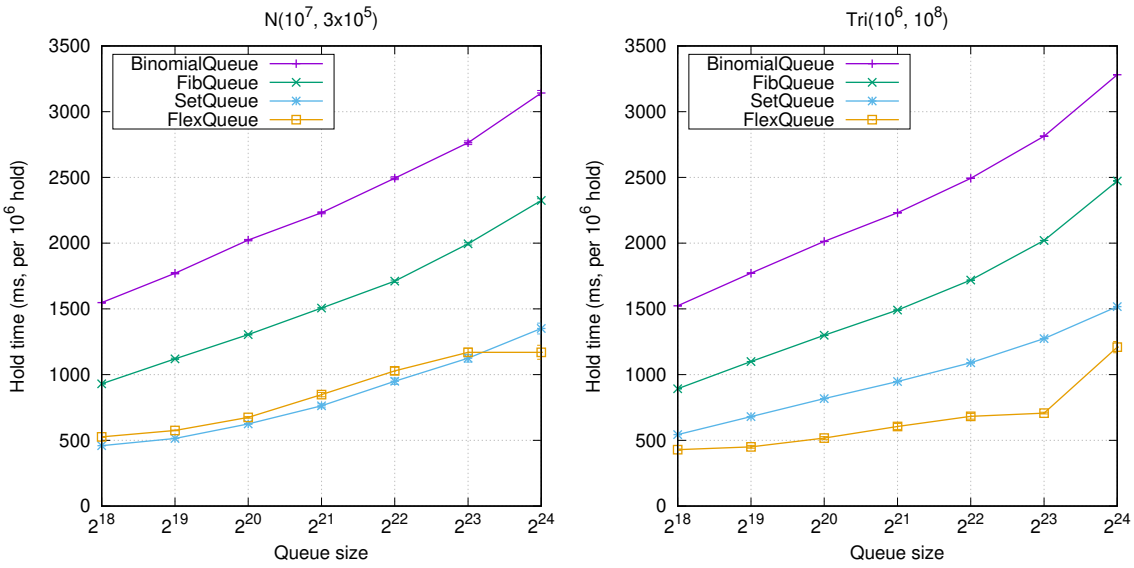


Figure 4.7: FlexQueue vs competitors

4.3.5 Other Distributions

Previous benchmarks employs a basic piece-wise uniform distribution that is unrealistic for a real workload. This section compares FlexQueue against competing implementations under synthetic, yet more typical of real world, input distributions. These include normal and triangular distributions.

Notably, in Figure 4.7, set queue performs almost identical to FlexQueue. This is because a normal distribution with a small variance is used. As such, a majority of the events are centred around the mean, therefore, it becomes difficult to choose a value of k such that $2k$ standard deviations of events can be distributed across all buckets evenly when the queue size is only slightly larger than the number of buckets. This reduces the effectiveness of the pointer array and increases the average latency of accessing A. It is therefore expected that FlexQueue should have better performance if the input distribution has several clumps rather than just one.

The triangular distribution in Figure 4.7 is similar to the result of adding multiple normal distributions with unique means. Such a distribution results in multiple, closely-spaced clusters of values rather than just a single cluster, and gives FlexQueue the opportunity to make use of the pointer array over a much larger range of events.

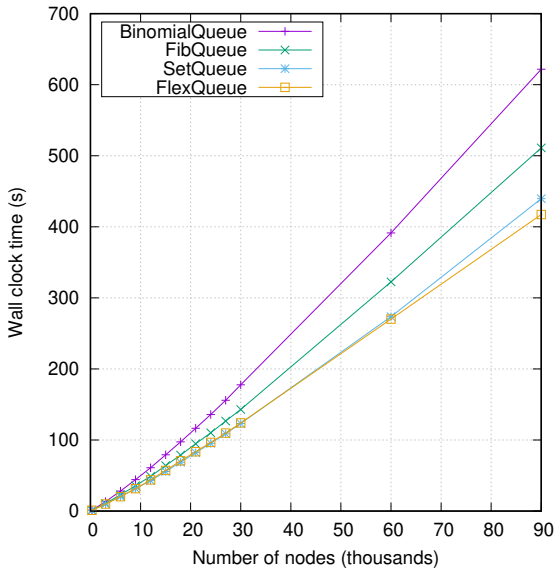


Figure 4.8: `ripng-network` topology using different scheduler

4.4 `ns3` Application

`ns3` [1] is an open-source, discrete event network simulator. It is particularly good at packet level simulation and creating models that are realistic enough that simulation results are considered very realistic. According to its documentation, the priority queue used as the scheduler is an STL set queue, which is implemented using a red-black tree. FlexQueue works as a drop-in replacement for `ns3`'s scheduler with no changes to the source code.

Included in the source code repository are workloads that can be readily compiled and executed. One of these workloads, '`ripng-network`' creates a small topology with 6 nodes and 7 links between them. This workload is intended to simulate the operation of Routing Information Protocol next generation (RIPng) by repeatedly disconnecting and reconnecting one of the low-cost links between two subnets. This causes the protocol to re-route traffic through a different, high-cost link that is otherwise unused. By increasing the number of copies t of this topology created during simulation, it is possible to observe the performance of FlexQueue under various queue sizes and determine whether its performance benefits carry over to a real world application.

Figure 4.8 shows the running time of the `ripng-network` simulation. As expected, FlexQueue does not out-perform set queue at smaller queue sizes, and is noticeably slower at these parameters. Note that the smallest queue sizes here are much smaller than those

during benchmarks. This shows that FlexQueue does not introduce significant overhead, even under parameters that it is not optimized for. By increasing the number of nodes used in the simulation, the number of queued events increases, and approaches the range of values that FlexQueue excels at, as suggested by previous benchmarks. At $t = 300$, FlexQueue's simulation run time is roughly 2% faster than that of set queue. Using the `gprof` profiler, it is discovered that priority queue related operations such as `insert` and `deleteMin` account for roughly 10% of the entire `ripng-network` simulation run time. Thus, this result translates to roughly a 20% improvement over set queue for priority queue operations only, since all other components of the simulation remains identical.

Chapter 5

Conclusion and Future Work

This thesis presents FlexQueue, a priority queue design that is suitable for use in system software such as discrete event simulators and operating system kernels. FlexQueue is different from other calendar queue variations in three ways. First, FlexQueue uses an efficient hierarchical bit vector to locate the bucket where the highest priority item is stored. This significantly lowers the cost of both `insert` and `deleteMin` operations. Searching through the bit vector still takes time proportional to the logarithm of the total number of buckets, but given that the number of buckets is fixed, the cost is practically constant. Second, in order to ensure the bit vector can be queried at all times, events that do not fit in the current horizon are stored separately in an overflow list L. Finally, to minimize the possibility of having too many events in L as a result of a skewed input distribution, FlexQueue uses a dynamic horizon that is able to track basic shifts in input characteristics.

A bench-marking tool has been implemented to evaluate the effectiveness of these changes. Since the bit vector is intended to operate on top of the pointer array, its cost is compared with operations on a linked list. This represents the cheapest operation possible on a given bucket, thus exaggerates the overhead of the bit vector, if any. Results show that in the worst case, accessing the bit vector is slightly faster than inserting elements into a linked list. In addition, because the implementation of the bit vector uses `bsf` whose input is at most one machine word, this limits the possibility of using a fanout factor larger than 64 bits. Benchmark experiments also demonstrate the effect of dynamic horizon when both queue size and distribution changes. FlexQueue's resizing policy is able to sense and track when the timestamps of the incoming events are increasing, and responds by modifying the lower and upper bound of the current horizon. Through a combination of these techniques, FlexQueue is able to perform competitively against popular tree-based priority queue implementations when the queue size is sufficiently large.

This thesis compares FlexQueue only with other tree-based implementations. A more general and comprehensive study should include selected list-based implementations mentioned in Chapter 2. In addition, a clear weakness of FlexQueue is that it is slightly slower when the queue size is small. In this scenario, most of the buckets are presumably empty. As a result, attempting to divide the already small universe of elements into a much larger number of buckets becomes more expensive than a simpler data structure such as a binary tree. A possible solution is to allow the number of buckets to decrease, if it is detected that a majority of buckets are unused. These ideas are left for future exploration.

References

[1] Ns-3. https://www.nsnam.org/.

[2] Jong Suk Ahn and Peter B. Danzig. Packet Network Simulation: Speedup and Accuracy Versus Timing Granularity. *IEEE/ACM Trans. Netw.*, 4(5):743–757, October 1996.

[3] Eyad Alkassar, Ernie Cohen, Mark Hillebrand, Mikhail Kovalev, and Wolfgang J. Paul. Verifying Shadow Page Table Algorithms. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, pages 267–270, Austin, TX, 2010. FMCAD Inc.

[4] A. Andersson and S. Nilsson. A New Efficient Radix Sort. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 714–721, November 1994.

[5] Mohit Aron and Peter Druschel. Soft Timers: Efficient Microsecond Software Timer Support for Network Processing. *ACM Trans. Comput. Syst.*, 18(3):197–228, August 2000.

[6] Eduard Babulak and Ming Wang. Discrete event simulation: State of the art. *International Journal of Online Engineering (iJOE)*, 4(2):60–63, 2007.

[7] H.A. Bahr and R.F. DeMara. Smart Priority Queue Algorithms for Self-Optimizing Event Storage. *Simulation Modelling Practice & Theory*, 12(1):15–40, 2004.

[8] R. Barkley and T. Lee. A Lazy Buddy System Bounded by Two Coalescing Delays. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 167–176, New York, NY, USA, 1989. ACM.

[9] Tim Bell and Bengt Aspvall. Sorting Algorithms as Special Cases of a Priority Queue Sort. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, pages 123–128, New York, NY, USA, 2011. ACM.

[10] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill. Translation Lookaside Buffer Consistency: A Software Approach. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS III, pages 113–122, New York, NY, USA, 1989. ACM.

[11] John H. Blackstone, Jr., Gary L. Hogg, and Don T. Phillips. A Two-List Synchronization Procedure for Discrete Event Simulation. *Commun. ACM*, 24(12):825–829, December 1981.

[12] J. Blazewicz, M. Drabowski, and J. Weglarz. Scheduling Multiprocessor Tasks to Minimize Schedule Length. *IEEE Transactions on Computers*, C-35(5):389–393, May 1986.

[13] M. Blum, R.W. Floyd, V. Pratt, R.L. Rivest, and R.E. Tarjan. Time Bounds for Selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[14] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and Implementation of an Efficient Priority Queue. *Mathematical systems theory*, 10(1):99–127, December 1976.

[15] B. B. Brandenburg and J. H. Anderson. On the Implementation of Global Real-Time Schedulers. In *2009 30th IEEE Real-Time Systems Symposium*, pages 214–224, December 2009.

[16] Romain Brette, Michelle Rudolph, Ted Carnevale, Michael Hines, David Beeman, James M. Bower, Markus Diesmann, Abigail Morrison, Philip H. Goodman, Frederick C. Harris, Milind Zirpe, Thomas Natschläger, Dejan Pecevski, Bard Ermentrout, Mikael Djurfeldt, Anders Lansner, Olivier Rochel, Thierry Vieville, Eilif Muller, Andrew P. Davison, Sami El Boustani, and Alain Destexhe. Simulation of networks of spiking neurons: A review of tools and strategies. *Journal of Computational Neuroscience*, 23(3):349–398, December 2007.

[17] Gerth Stølting Brodal. A Survey on Priority Queues. In Andrej Brodnik, Alejandro López-Ortiz, Venkatesh Raman, and Alfredo Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms*, number 8066 in Lecture Notes in Computer Science, pages 150–163. Springer Berlin Heidelberg, 2013.

[18] Mark Robbin Brown. The Analysis of a Practical and Nearly Optimal Priority Queue. Technical report, Stanford University, Stanford, CA, USA, 1977.

[19] R. Brown. Calendar Queues: A Fast 0(1) Priority Queue Implementation for the Simulation Event Set Problem. *Commun. ACM*, 31(10):1220–1227, October 1988.

[20] Svante Carlsson, J. Ian Munro, and Patricio V. Poblete. An Implicit Binomial Queue with Constant Insertion Time. In *SWAT 88: 1st Scandinavian Workshop on Algorithm Theory Halmstad, Sweden, July 5–8, 1988 Proceedings*, pages 1–13. Springer, Berlin, Heidelberg, July 1988.

[21] Yuhua Chen, Jonathan S. Turner, and Pu-Fan Mo. Optimal Burst Scheduling in Optical Burst Switched Networks. *Journal of Lightwave Technology*, 25(8):1883–1894, August 2007.

[22] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. RadixVM: Scalable Address Spaces for Multithreaded Applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 211–224, New York, NY, USA, 2013. ACM.

[23] John Comfort. The Simulation of a Master-Slave Event Set Processor. *Simulation*, 42(3):117–124, 1984.

[24] Adam M. Costello and George Varghese. Redesigning the BSD timer facilities. *Software: Practice and Experience*, 28(8):883–896, July 1998.

[25] Clark Allan Crane. *Linear Lists and Priority Queues As Balanced Binary Trees*. PhD thesis, Stanford University, Stanford, CA, USA, 1972.

[26] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[27] Robert W. Floyd. Algorithm 113: Treesort. *Commun. ACM*, 5(8):434–, August 1962.

[28] W. R. Franta and Kurt Maly. An Efficient Data Structure for the Simulation Event Set. *Commun. ACM*, 20(8):596–602, August 1977.

[29] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The Pairing Heap: A New Form of Self-Adjusting Heap. *Algorithmica*, 1(1-4):111–129, November 1986.

[30] Michael L. Fredman and Robert Endre Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *J. ACM*, 34(3):596–615, July 1987.

[31] Jakob Gruber, Jesper Larsson Träff, and Martin Wimmer. Benchmarking Concurrent Priority Queues: Performance of k-LSM and Related Data Structures. *arXiv:1603.05047 [cs]*, March 2016.

[32] David R. Hanson. Fast Allocation and Deallocation of Memory Based on Object Lifetimes. *Software: Practice and Experience*, 20(1):5–12, January 1990.

[33] E.S.H. Hou, N. Ansari, and Hong Ren. A Genetic Algorithm for Multiprocessor Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 5(2):113–120, February 1994.

[34] Shafagh Jafer, Qi Liu, and Gabriel Wainer. Synchronization Methods in Parallel and Distributed Discrete-Event Simulation. *Simulation Modelling Practice and Theory*, 30:54–73, 2013.

[35] Donald B. Johnson. A Priority Queue in Which Initialization and Queue Operations Takeo(loglogd) Time. *Mathematical systems theory*, 15(1):295–309, December 1981.

[36] Douglas W. Jones. An Empirical Comparison of Priority-Queue and Event-Set Implementations. *Commun. ACM*, 29(4):300–311, April 1986.

[37] J. Kazmier Leonard. *Schaum's Outline of Business Statistics*. New York: McGraw-Hill Professional, 2009.

[38] Saverio Mascolo, Claudio Casetti, Mario Gerla, M. Y. Sanadidi, and Ren Wang. TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, MobiCom '01, pages 287–297, New York, NY, USA, 2001. ACM.

[39] K. McLaughlin, S. Sezer, H. Blume, X. Yang, F. Kupzog, and T. Noll. A Scalable Packet Sorting Circuit for High-Speed WFQ Packet Scheduling. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(7):781–791, July 2008.

[40] SeungHyun Oh and JongSuk Ahn. Dynamic Calendar Queue. In *Proceedings 32nd Annual Simulation Symposium*, pages 20–25, 1999.

[41] V. Paxson, M. Allman, J. Chu, and M. Sargent. Computing TCP's Retransmission Timer. RFC 6298, RFC Editor, June 2011.

[42] Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. 30 Seconds is Not Enough!: A Study of Operating System Timer Usage. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 205–218, New York, NY, USA, 2008. ACM.

[43] Rajeev Raman. A Summary of Shortest Path Results. *Kings College (London) TR*, pages 96–13, 1996.

[44] J. Riboe. Improvement of the Calendar Queue Algorithm. *Under elaboration at Dept. of Telecommunication and Computer Systems, The Royal Inst. of Technology, Stockholm*, 1990.

[45] Robert Rönngren and Rassul Ayani. A Comparative Study of Parallel and Sequential Priority Queue Algorithms. *ACM Trans. Model. Comput. Simul.*, 7(2):157–209, April 1997.

[46] Robert Rönngren, Jens Riboe, and Rassul Ayani. Lazy Queue: An Efficient Implementation of the Pending-event Set. In *Proceedings of the 24th Annual Symposium on Simulation*, ANSS '91, pages 194–204, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[47] Sheldon M. Ross. *Introduction to Probability and Statistics for Engineers and Scientists*. Elsevier/Academic Press, Amsterdam; Boston, 3rd edition, 2004.

[48] Suresh Siddha, Venkatesh Pallipadi, and AVD Ven. Getting Maximum Mileage Out of Tickless. In *Proceedings of the Linux Symposium*, volume 2, pages 201–207. Citeseer, 2007.

[49] Rick Siow, Mong Goh, Ian Li, and Jin Thng. DSplay: An Efficient Dynamic Priority Queue Structure For Discrete Event Simulation. In *Proceedings of the SimTecT Simulation Technology and Training Conference*, Canberra, Australia, 2004.

[50] D. Sleator and R. Tarjan. Self-Adjusting Heaps. *SIAM Journal on Computing*, 15(1):52–69, February 1986.

[51] Sun, Xianda. *Concurrent High-Performance Persistent Hash Table In Java*. Master's Thesis, UWSpace, 2015.

[52] Kah Leong Tan and Li-Jin Thng. SNOOPy Calendar Queue. In *Proceedings of the 32Nd Conference on Winter Simulation*, WSC '00, pages 487–495, San Diego, CA, USA, 2000. Society for Computer Simulation International.

[53] Robert Endre Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.

[54] Mikkel Thorup. On RAM Priority Queues. *SIAM Journal on Computing; Philadelphia*, 30(1):24, 2000.

[55] Mikkel Thorup. Integer Priority Queues with Decrease Key in Constant Time and the Single Source Shortest Paths Problem. In *Proceedings of the Thirty-Fifth Annual*

ACM Symposium on Theory of Computing, STOC '03, pages 149–158, New York, NY, USA, 2003. ACM.

[56] T. N. Van, V. T. Thien, S. N. Kim, N. P. Ngoc, and T. N. Huu. A High Throughput Pipelined Hardware Architecture for Tag Sorting in Packet Fair Queuing Schedulers. In *2015 International Conference on Communications, Management and Telecommunications (ComManTel)*, pages 41–45, December 2015.

[57] G. Varghese and T. Lauck. Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, pages 25–38, New York, NY, USA, 1987. ACM.

[58] Jean G. Vaucher and Pierre Duval. A Comparison of Simulation Event List Algorithms. *Commun. ACM*, 18(4):223–230, April 1975.

[59] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal. DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory. In *2011 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 340–349, October 2011.

[60] Jean Vuillemin. A Data Structure for Manipulating Priority Queues. *Commun. ACM*, 21(4):309–315, April 1978.

[61] H. Wang and B. Lin. Per-Flow Queue Management with Succinct Priority Indexing Structures for High Speed Packet Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 24(7):1380–1389, July 2013.

[62] JW J. Williams. Algorithm 232: Heapsort. *Commun. ACM*, 7:347–348, 1964.

[63] C. S. Wong, I. K. T. Tan, R. D. Kumari, J. W. Lam, and W. Fun. Fairness and interactive performance of O(1) and CFS Linux kernel schedulers. In *2008 International Symposium on Information Technology*, volume 4, pages 1–8, August 2008.

[64] Guanhua Yan and S. Eidenbenz. Sluggish Calendar Queues for Network Simulation. In *And Simulation 14th IEEE International Symposium on Modeling, Analysis*, pages 127–136, September 2006.

[65] Liron Yatziv, Alberto Bartesaghi, and Guillermo Sapiro. O(n) Implementation of the Fast Marching Algorithm. *Journal of Computational Physics*, 212(2):393–399, March 2006.